

TODO

- P56, 2.7.4, 似乎是改动的快排;
- P67, 3.3.7, 表达式转换;
- P130, 11; BST和DST;
- P130, 15; 看了解析才会;
- P160~162, 搜索算法;
- BST iter;

1 导论

1.4 Python基础

`range(a, b)` 和 `list[a:b]` 差不多, 都从a下标开始, 但是不包括b下标。

2 算法分析

2.3 Python数据结构的性能

- `list = list + [i]` 比 `list.append(i)` 慢得多, 连接比追加慢;
- `pop()` 和 `pop(i)` 时间复杂度分别为1和n, 即使是 `pop(0)` 也要慢很多;

表 2-2 Python 列表操作的大 O 效率

操作	大O效率
索引	$O(1)$
索引赋值	$O(1)$
追加	$O(1)$
<code>pop()</code>	$O(1)$
<code>pop(i)</code>	$O(n)$
<code>insert(i, item)</code>	$O(n)$
删除	$O(n)$
遍历	$O(n)$
包含	$O(n)$
切片	$O(k)$
删除切片	$O(n)$
设置切片	$O(n+k)$
反转	$O(n)$
连接	$O(k)$
排序	$O(n \log n)$
乘法	$O(nk)$

表 2-3 Python 字典操作的大 O 效率

操作	大 O 效率
复制	$O(n)$
取值	$O(1)$
赋值	$O(1)$
删除	$O(1)$
包含	$O(1)$
遍历	$O(n)$

3 基本数据结构

3.3 栈

- 栈具有反转特性；
- 在py里可以自定义一个类，用原生的list实现各种方法；
-

表 3-1 栈操作示例

栈 操 作	栈 内 容	返 回 值
s.isEmpty()	[]	True
s.push(4)	[4]	
s.push('dog')	[4, 'dog']	
s.peek()	[4, 'dog']	'dog'
s.push(True)	[4, 'dog', True]	
s.size()	[4, 'dog', True]	3
s.isEmpty()	[4, 'dog', True]	False
s.push(8.4)	[4, 'dog', True, 8.4]	
s.pop()	[4, 'dog', True]	8.4
s.pop()	[4, 'dog']	True
s.size()	[4, 'dog']	2

3.4 队列

- 和栈相反，先进先出，FIFO；
- 主要内容就是enqueue和enqueue；
- 可以用list实现；

表 3-5 队列操作示例

队列操作	队列内容	返 回 值
q.isEmpty()	[]	True
q.enqueue(4)	[4]	
q.enqueue('dog')	['dog', 4]	
q.enqueue(True)	[True, 'dog', 4]	
q.size()	[True, 'dog', 4]	3
q.isEmpty()	[True, 'dog', 4]	False
q.enqueue(8.4)	[8.4, True, 'dog', 4]	
q.dequeue()	[8.4, True, 'dog']	4
q.dequeue()	[8.4, True]	'dog'
q.size()	[8.4, True]	2

3.5 双端队列（deque）

- 像是stack和queue的结合，从两端都可以添加或者移除元素；

- add、remove和front、rear四种结合，以及查询个数和是否为空；
- 这里会涉及到 `pop(0)` 和 `append(0, i)`，可以把这两种都放在前端，这样统一从后端比较快；

3.6.2 无序列表：链表

- 首先需要一个Node类，包括data和next两种属性；
- 重点在于Unorderlist类；
 - head用来保存head；
 - isEmpty看有没有节点；
 - add，添加到开头比较方便；
 - 遍历：
 - length
 - search
 - remove：需要双指针
 - append
 - insert
 - index
 - pop

3.6.3 有序列表

比如一些数字，它们会按照数字大小排列，而不是直接添加到开头——元素的位置取决于他们的特征。

4 Recursion

4.2 Introduction

The Three Principles of Recursion

以递归求数列的和为例：

1. 递归算法必须有基本情况（数列长度为1时返回）；
2. 递归算法必须改变其状态并向基本情况靠近（数列拆分为**首项**和**其余项**）；
3. 递归算法必须递归地调用自己（返回**首项**和**其余项的和**）。

4.3 Realization

递归通过**栈帧**来实现。

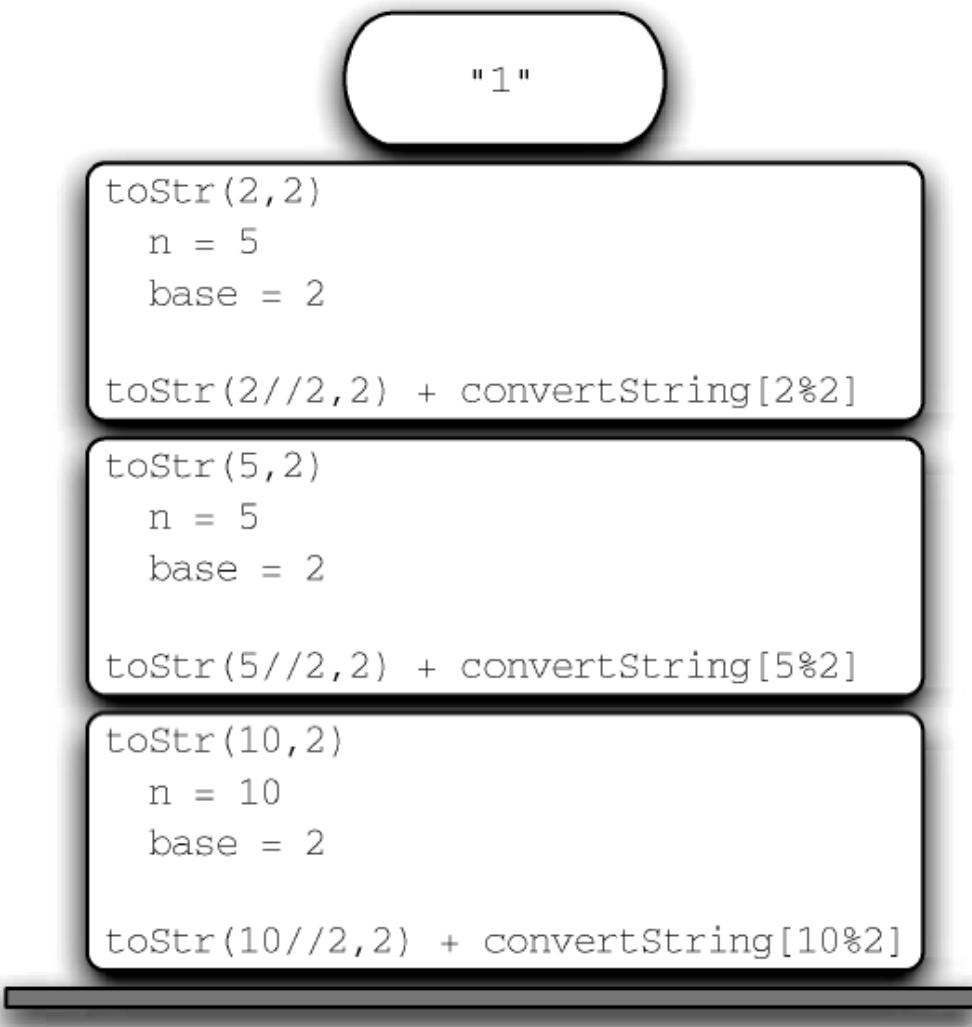


图 4-6 调用栈示例

栈底部的将保留函数和使用的变量（局部变量作用域为该帧）；

栈顶计算的结果将一层层向下替换，直至栈空；

4.7 DP, GA

DP: Dynamic Programming

GA: Greedy Algorithm

例子：

有1、5、10、25美分的硬币，而我需要63美分，怎么才能让硬币数量最少呢？

GA：试图最大程度地解决问题，但这只是局部最优解。如果GA的话，需要2个25美分，1个10美分，3个1美分，暂时是全局最优；

但如果还有21美分的硬币，最优解是3个21美分硬币——这时候GA就不起作用了。

利用递归的思想：

1. 递归算法必须有基本情况（数列长度为1时返回）；

2. 递归算法必须改变其状态并向基本情况靠近（数列拆分为首项和其余项）；

3. 递归算法必须递归地调用自己（返回首项和其余项的和）。

1. 基本情况：当剩下的金额和某个硬币金额一样；

2. 多种靠近方式：

1. 1美分+剩下的；
 2. 5美分+剩下的；
 3. 10美分+剩下的；

3. 递归：拆分成一个硬币剩下的金额；

但这有很大的问题：

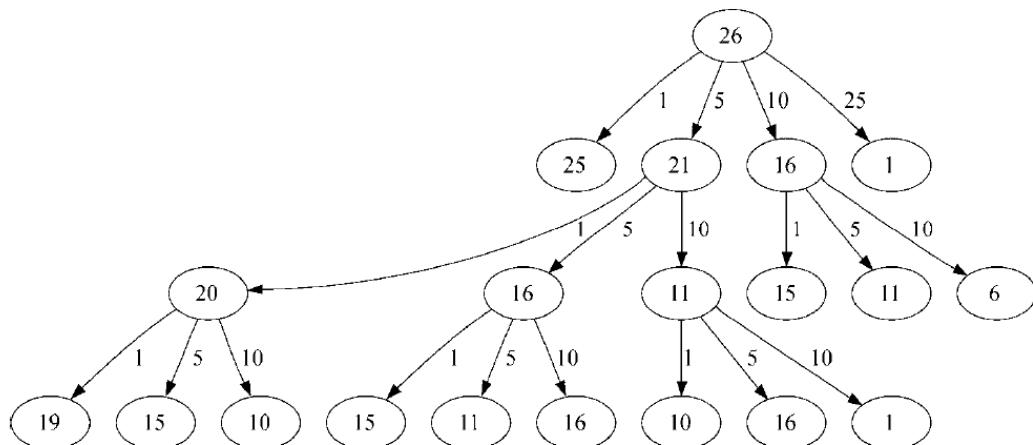


图 4-14 递归调用树

重复调用的次数太多，比如16就出现了很多次

初始解法：

```
1 def calc_coins_recursion(self, val_list: list, charge: int):
2     if charge in val_list:
3         return 1
4     else:
5         return 1 + min(self.calc_coins_recursion(val_list, charge - x) for x
in val_list if x < charge)
```

cache 解法，速度快了很多：

```
1 self.cache = dict()
2
3 def calc_coins_recursion_cache(self, val_list: list, charge: int):
4     if _ := self.cache.get(charge):
5         return _
6     else:
7         pass
8
9     if charge in val_list:
10        tmp = 1
11    else:
12        tmp = 1 + min(self.calc_coins_recursion_cache(val_list, charge - x)
13 for x in val_list if x < charge)
```

```
14     self.cache[charge] = tmp  
15     return tmp
```

dp解法：

```
1 def dp_coins(self, val_list, change):  
2     # build  
3     for i in range(1, change+1):  
4         if i in val_list:  
5             self.dp[i] = 1  
6         else:  
7             self.dp[i] = min((self.dp[i-x] + 1 for x in val_list if x <  
8 i))  
9     return self.dp[change]
```

DP是自下而上的，从初始值开始构建整个表格；虽然占用了一些空间用于储存，但是速度很快
递归是自上而下的，有可能会栈溢出

5 Search & Sort

5.2 搜索

5.2.1 顺序搜索

在有序和无序列表中搜索元素，区别在于：有序的蕴含信息，超过某个元素后，剩下的元素也不可能再是需要的元素。

5.2.2 二分搜索

前提：有序；

并且要考虑排序的成本有多高；

用两个 index 比 slice 要快，因为 slice 是 $O(n)$ 。

5.2.3 散列

核心：散列函数，处理冲突，再散列

散列，散列函数

散列中的每个位置被称为 Slot，假设是从 0 到 $m-1$ 共 m 个；每个元素经过散列函数计算后得到介于 0 和 $m-1$ 的数值，就可以插入槽——因此第一个简单的想法就是取余，对 m 取余。

尽管初始大小可以任意指定，但选用一个素数很重要，这样做可以尽可能地提高冲突处理算法的效率。

$$\text{hash}(x) = x \bmod m$$

但这个函数不完美，如果两个数计算得到的余数一样，就会发生冲突；我们希望对不同的值进行运算能得到各不同的值——但完美散列函数并不存在。

- **折叠法：**比如有个电话号码，每3位成为一组，加起来（或者每隔一组数先反转一次，如256变成652再相加）得到一个数，然后再取余；

- **平方取中法**: 先求元素的平方，取中间的几位数再取余；
- **如果是字符串**: 可以用 ascii 值计算；为了避免 dog 和 god 这种异序词产生相同的值，还可以根据位置加上权重；

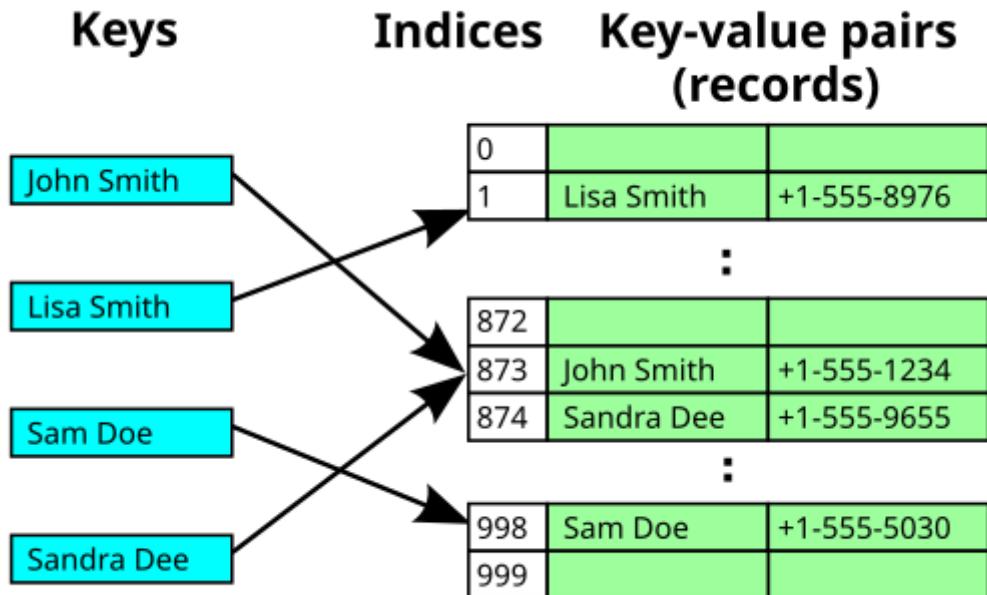
散列函数也不能太复杂，不然开销太大，有可能抵消 $O(n)$ 的优势！

冲突：接受不完美

假如还是之前的例子，77 会放在 Slot 0 里，但假如再来一个 55，就会出现冲突。

Open Addressing, Closed Hashing, 开放定址法:

- **线性探测, Linear Probing**: 冲突的时候，从头开始遍历；发现 Slot 1 是空的，就把 55 放进去；



但这会产生**聚集现象**，如果一个槽发生太多冲突，线性探测会填满其附近的槽，而这会影响到后续插入的元素。

0	1	2	3	4	5	6	7	8	9	10
77	44	55	20	26	93	17	None	None	31	54

图 5-9 散列值为 0 的元素聚集在一起

为了避免聚集，可以不每次 +1，而是每次 +3；但这都是线性的：这就需要一个**再散列函数 (Rehashing Function)** 来计算新的 Slot。

- **平方探测, Quadratic Probing**:

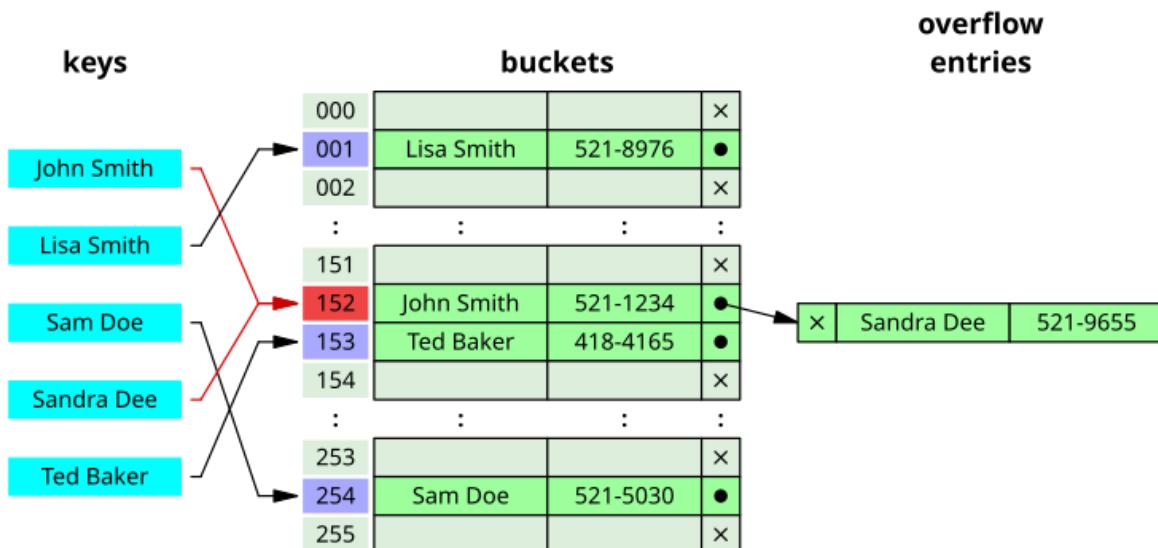
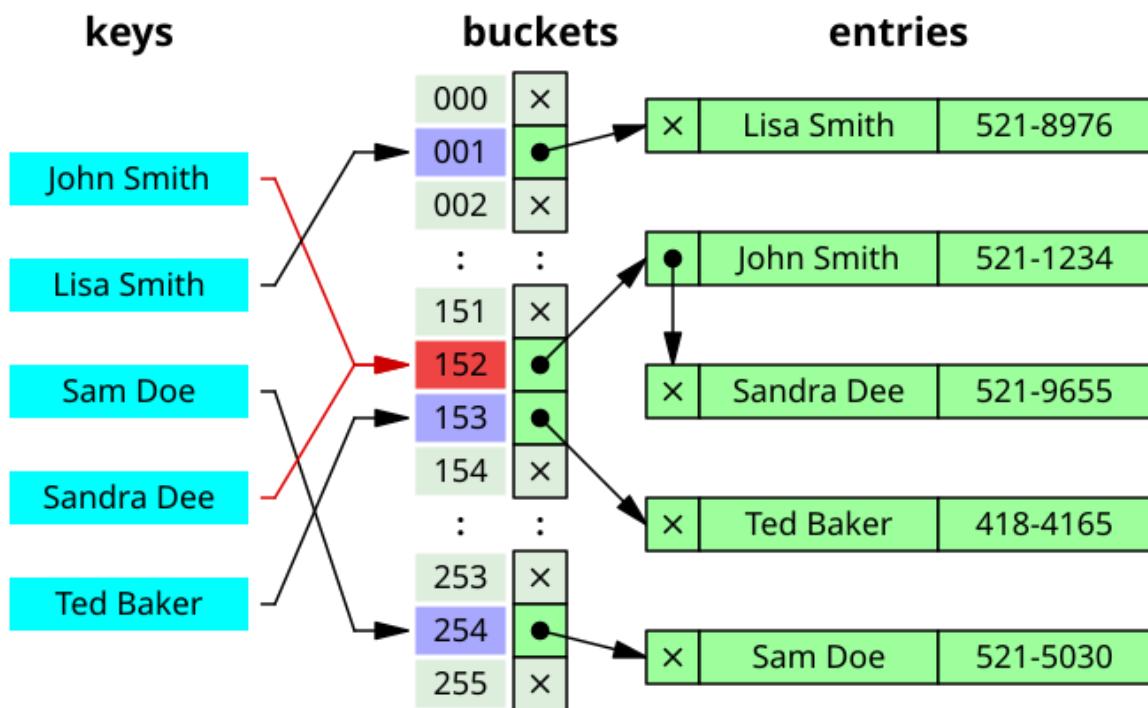
用二次函数计算：一开始 +1，后来 +4，然后 +9

- **双重哈希, Double Hashing**:

用另一个 hash 计算。

- **链接法, Separate Chaining**:

把 Slot 改造成一个链表。



实现

[Python 中 Dict 的实现](#)

字典是最有用的 Python 集合之一。第 1 章说过，字典是存储键–值对的数据类型。键用来查找关联的值，这个概念常常被称作映射。

映射抽象数据类型定义如下。它是将键和值关联起来的无序集合，其中的键是不重复的，键和值之间是一一对应的关系。映射支持以下操作。

- `Map()` 创建一个空的映射，它返回一个空的映射集合。

- ❑ `put(key, val)` 往映射中加入一个新的键–值对。如果键已经存在，就用新值替换旧值。
- ❑ `get(key)` 返回 `key` 对应的值。如果 `key` 不存在，则返回 `None`。
- ❑ `del` 通过 `del map[key]` 这样的语句从映射中删除键–值对。
- ❑ `len()` 返回映射中存储的键–值对的数目。
- ❑ `in` 通过 `key in map` 这样的语句，在键存在时返回 `True`，否则返回 `False`。

可以用两个列表实现，一对k-v有相同的index。

时间复杂度，载荷因子

载荷因子：

$$\lambda = \frac{\text{Elements Count}}{\text{Hash Table Size}}$$

载荷因子小，则不容易冲突；否则找到新的 slot 就更难，或者链表就更长。

和之前一样，来看看搜索成功和搜索失败的情况。采用线性探测策略的开放定址法，搜索成功的平均比较次数如下。

$$\frac{1}{2} \left(1 + \frac{1}{1 - \lambda} \right)$$

搜索失败的平均比较次数如下。

$$\frac{1}{2} \left[1 + \left(\frac{1}{1 - \lambda} \right)^2 \right]$$

若采用链接法，则搜索成功的平均比较次数如下。

$$1 + \frac{\lambda}{2}$$

搜索失败时，平均比较次数就是 λ 。

5.3 排序(Sorting Algorithm)

5.3.1 冒泡排序(Bubble Sort)

- 第一轮：对于前 $n-1$ 个元素，比较他们和后一个的顺序；
- 第一轮结束后，最大/最小的元素已经在末尾了；
- 第 i 轮：对于前 $n-i$ 个元素，比较；
- 到 $n-1$ 轮结束后，还剩一个元素就在他应该的位置上；
- $O(n^2)$ 。

第一轮遍历

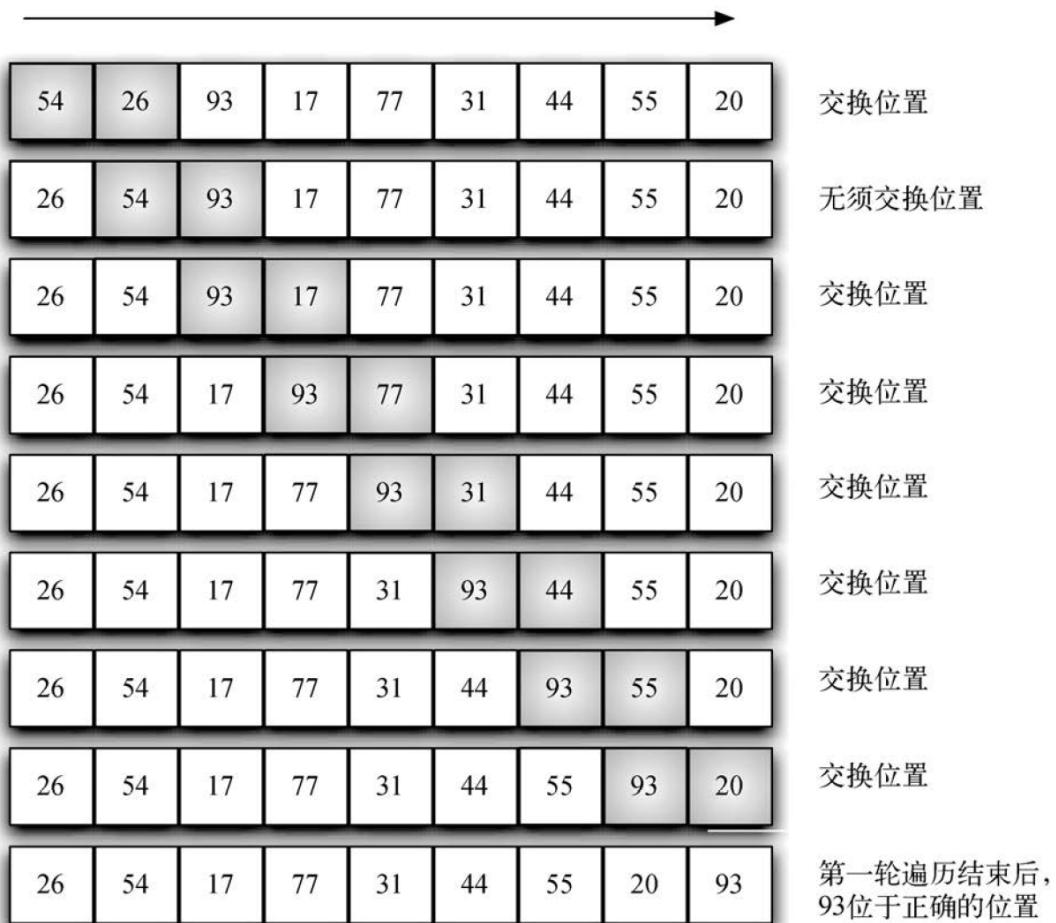


图 5-13 冒泡排序的第一轮遍历过程

改进方法：如果某一轮时，没有进行交换，说明可以提前结束。

5.3.2 选择排序(Selection Sort)

1. 遍历全部，把最大的放在最后；
2. 遍历前 $n-1$ 个，把最大的放在 $n-1$ 的位置；
3.
4. 遍历前两个。

选择排序的时间复杂度仍然是 $O(n^2)$ ，因为找到最大的值的索引这仍然是个 $O(n)$ 的问题。

但是交换的次数相比冒泡排序要少。

5.3.3 插入排序(Insertion Sort)

在列表较低的一端维护一个有序的子列表，并逐个将每个新元素“插入”这个子列表。

时间复杂度仍然是 $O(n^2)$ 。

这里的插入涉及到元素位置的移动（实质上为赋值），比交换要快：

移动操作和交换操作有一个重要的不同点。总体来说，交换操作的处理时间大约是移动操作的 3 倍，因为后者只需进行一次赋值。在基准测试中，插入排序算法的性能很不错。

54 26 93 17 77 31 44 55 20	将54视作只含单个元素的有序子列表
26 54 93 17 77 31 44 55 20	插入26
26 54 93 17 77 31 44 55 20	插入93
17 26 54 93 77 31 44 55 20	插入17
17 26 54 77 93 31 44 55 20	插入77
17 26 31 54 77 93 44 55 20	插入31
17 26 31 44 54 77 93 55 20	插入44
17 26 31 44 54 55 77 93 20	插入55
17 20 26 31 44 54 55 77 93	插入20

图 5-16 插入排序

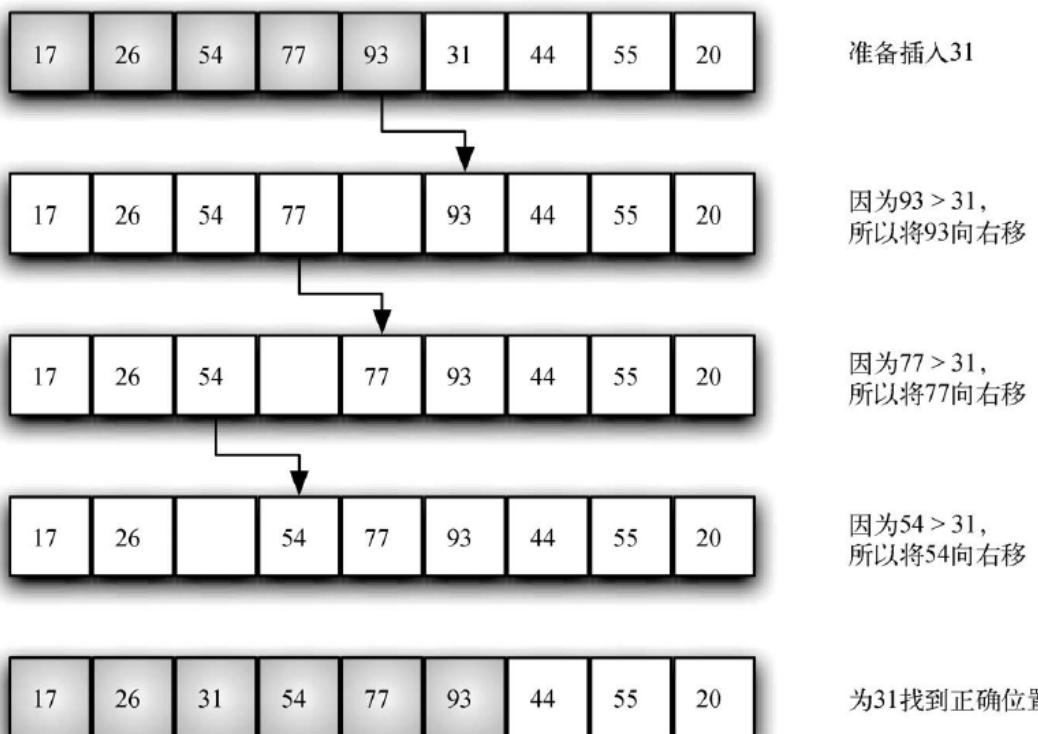


图 5-17 插入排序的第 5 轮遍历

5.3.4 希尔排序(Shell Sort)

基本排序算法中的插入排序虽然逻辑较为简单，但当排序规模较大时，经常需要将元素一位一位地从一端移动到另一端，效率非常低。于是Donald Shell设计了一种基于插入排序的改进版排序算法，故被命名为 Shell Sort。

希尔排序也称“递减增量排序”，它对插入排序做了改进，将列表分成数个子列表，并对每一个子列表应用插入排序。如何切分列表是希尔排序的关键——并不是连续切分，而是使用增量 i （有时称作步长）选取所有间隔为 i 的元素组成子列表。

其中一个通项公式，可根据数据规模计算各项步长。

$$\text{step}(n) = \frac{1}{2}(3^n - 1), (n \geq 1)$$

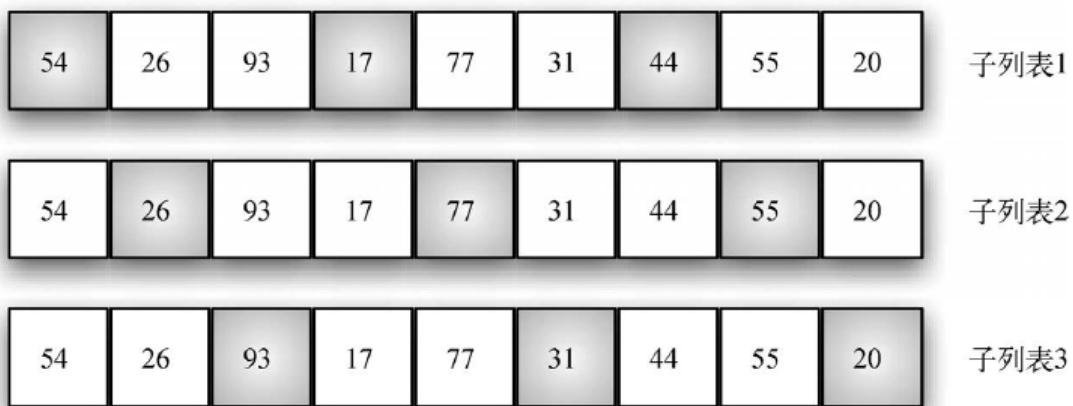


图 5-18 增量为 3 的希尔排序

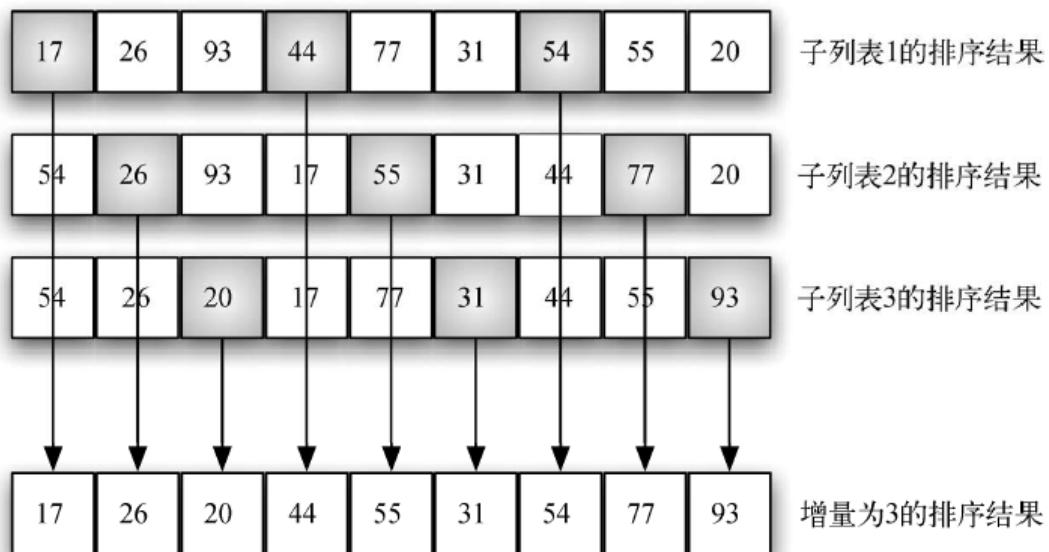


图 5-19 为每个子列表排序后的结果

尽管此时并不能说是完全有序，但是又前进了一步。

接下来减小增量，直到最后增量为1，相当于一次普通的 Insertion Sort。

希尔排序的时间复杂度大概介于 $O(n)$ 和 $O(n^2)$ 之间，取决于 step 序列。

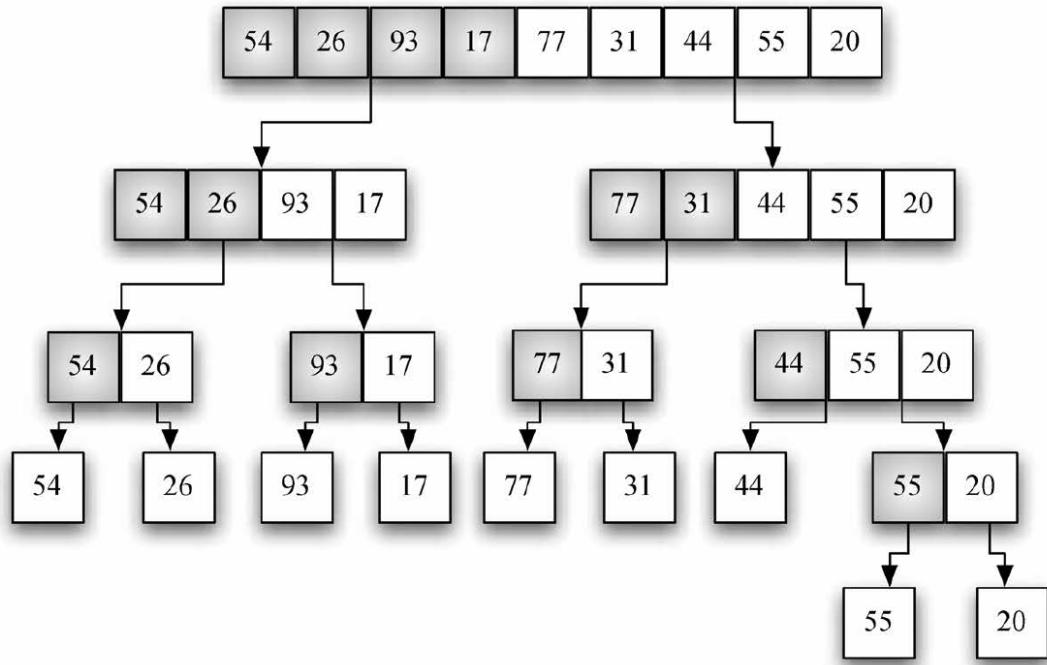
5.3.5 归并排序(Merge Sort)

递归地排序。

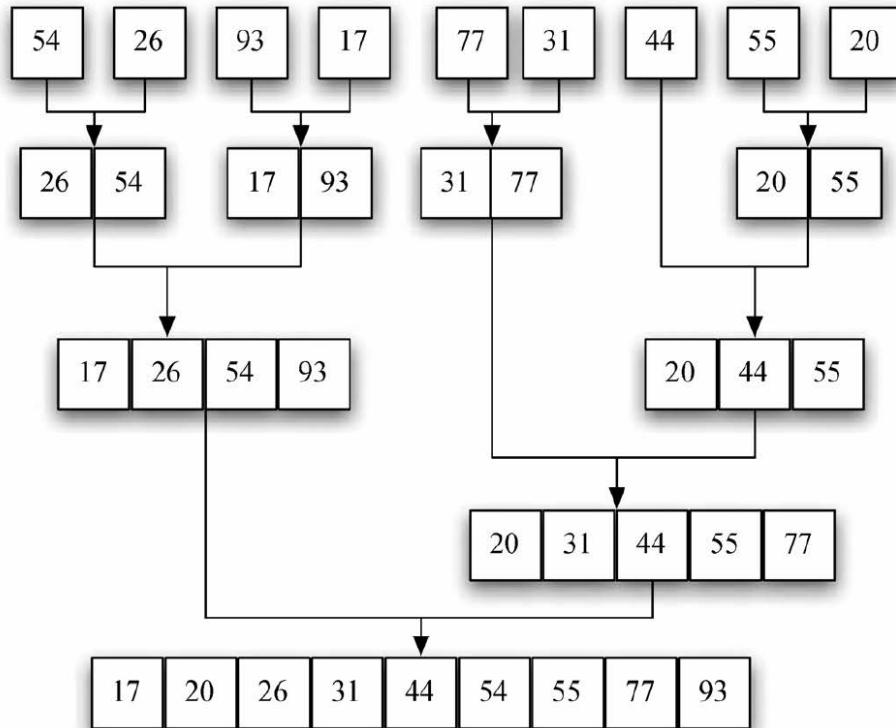
- 递归退出条件：列表长度为0或者1，此时有序；
- 列表长度长，就分成两部分；
- 合并：相当于合并两个有序列表。

拆分是 $O(\log n)$ ，每次归并是 $O(n)$ ，所以一共是 $O(n \log n)$ 。

但是切片是 $O(k)$ ，所以应该不用切片。



(a) 拆分



(b) 归并

图 5-22 归并排序中的拆分和归并

= 不用切片的话，得用一个其他的列表储存值，再还给原先的列表——但是尽管这么做，速度和用切片的没什么区别，甚至更慢——我不知道为什么。 ==

5.3.6 快速排序(Quick Sort)

也是递归和分治，但是不占用额外空间，直接更改。

快速排序算法首先选出一个**基准值**。尽管有很多种选法，但为简单起见，可以选取列表中的第一个元素。

基准值的作用是帮助切分列表（一轮排序后，小于基准值的在左边，大于基准值的在右边）。

在最终的有序列表中，基准值的位置通常被称作**分割点**，算法在分割点切分列表，以进行对快速排序的子调用。

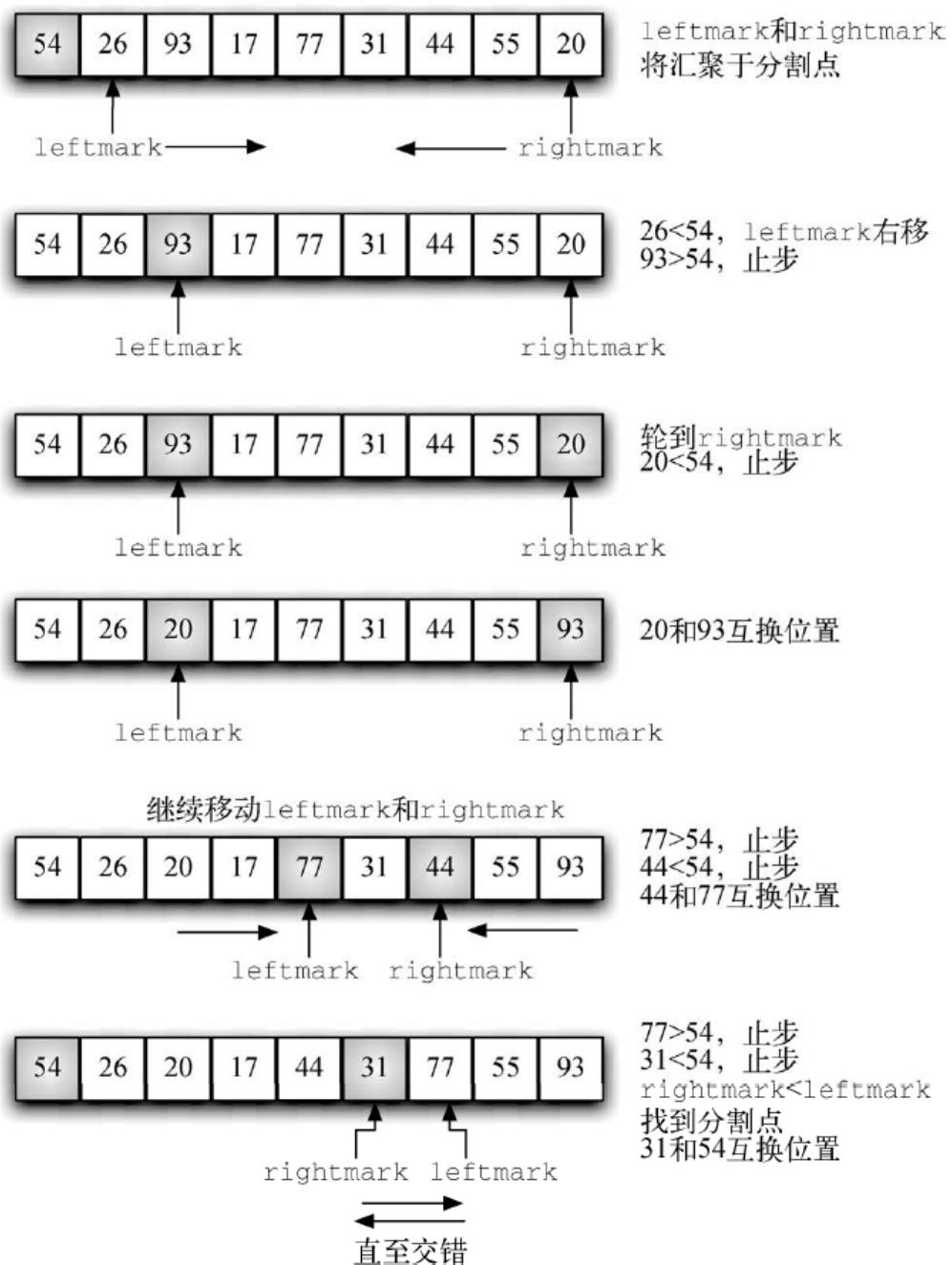


图 5-24 为 54 寻找正确位置

1. 确定一个基准值，未来将小于这个值的放在左边，大于的放在右边；

2. 为了方便，假设这个基准值就是列表首位的值；

3. 双指针，分别指向 [1] 和 [end]；

4. **大前提：右指针在左指针左边（重合也不能算）：**

1. 首先移动左指针，直到内容大于基准值停下；

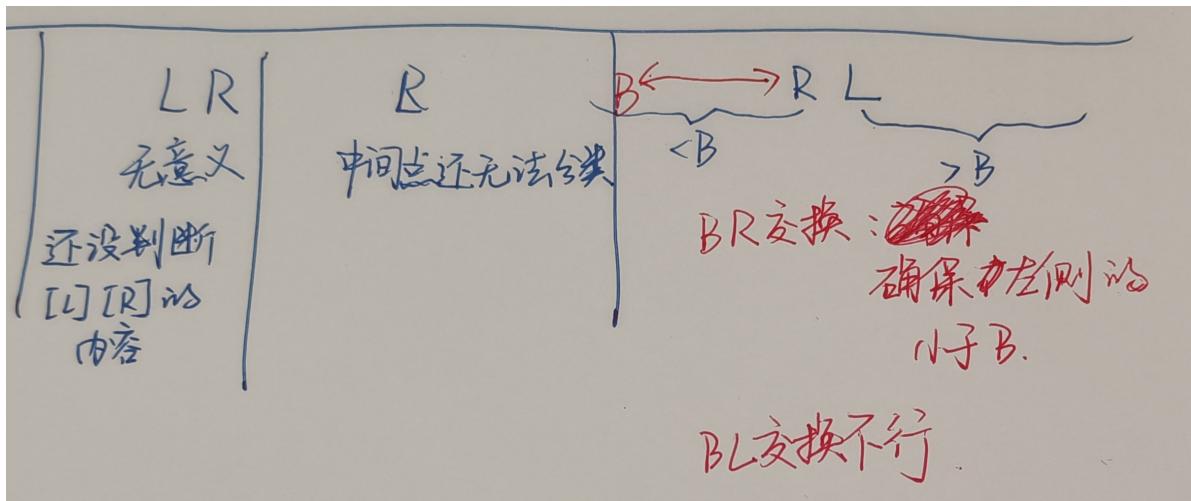
2. 再移动右指针，直到内容小于基准值停下；

3. 如果不满足大前提，就交换指针内容；

4. 继续循环；

5. 如果满足大前提（双指针刚好错位），此时左指针所在位置大于基准值，右指针所在位置小于基准值。现在交换右指针和基准值的位置，则从左指针开始的区域都大于基准值，交换前右指针前边的区域都小于基准值。

图解（在每次循环开始前）：



但这么做有一个问题：理想情况下，每次分割点都在中间，这样就是 $O(n \log n)$ 的；但假设这是一个有序序列 $[1, 2 \dots n]$ ，那每次都将分成 $[i, i+1 \dots n]$ 和一个空列表，时间复杂度也就退化成 $O(n^2)$ 。

一个改进的想法是**三数取中法**，从开始、中间和结尾三个数中选择一个中间的数作为基准值，因为这个数的实际位置更有希望靠近中间。

为了不破坏原有程序，将所选择的基准值和首位进行调换——这样之前的程序还可以接着用。

5.3.7 桶排序(Bucket Sort)

假设输入数据是整数，而且范围是已知的，比如100以内。可以根据 $0 \sim 9 | 10 \sim 19 \dots$ 把他们先分到不同的桶里；

再在每个桶里进行排序，比如用快排；

最后把每个桶的结果合并。

5.3.8 堆排序(Heap Sort)

用二叉堆进行排序。

先构建堆，再每次 pop 一个堆顶元素。

5.3.9 计数排序(Counting Sort)

假设输入的值的范围比较小，例如10以内的整数；统计每个数出现的次数，并按照次数再输出。

这个方法需要的空间较大。

5.3.10 基数排序(Radix Sort)

这个基数实际指的是进制的基数，比如十进制的基数是10。

假设都是正整数，而且已知最大的数有几位。

1. 把个位数是1/2...9/0 的放在一个桶里；
2. 按顺序取出来，重组；
3. 再把十位数是1/2...9/0 的放在一个桶里；
4. 取出来，按顺序重组；
5. 循环，直到最高位。

```
1 [601, 930, 890, 735, 628, 838, 301, 146, 306, 608, 187]
2 [930, 890, 601, 301, 735, 146, 306, 187, 628, 838, 608] # 按个位数
3 [601, 301, 306, 608, 628, 930, 735, 838, 146, 187, 890] # 按十位数
4 [146, 187, 301, 306, 601, 608, 628, 735, 838, 890, 930] # 按百位数
```

排序算法总结，优劣对比

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

✓ ✓ 测试结果	24分钟 26秒
✓ ✓ fuck_SortUtils	24分钟 26秒
✓ ✓ testcase_collection	24分钟 26秒
✓ test_bubbleSort	9分钟 39秒
✓ test_bucketSort	1秒 201毫秒
✓ test_builtinSort	65毫秒
✓ test_countingSort	55毫秒
✓ test_heapSort	2秒 797毫秒
✓ test_insertionSort	3分钟 20秒
✓ test_mergeSort	1秒 499毫秒
✓ test_mergeSort_Opt	1秒 503毫秒
✓ test_quicksort	1秒 93毫秒
✓ test_quicksort_Opt	1秒 320毫秒
✓ test_radixSort	847毫秒
✓ test_selectionSort	1分钟 19秒
✓ test_shellSort	23秒 946毫秒
✓ test_shortBubbleSort	9分钟 34秒

上图条件：只有正整数，30个，数据数量20000，数值范围1000以内

排序方法的选择

在随机数排序结果中，列表长度比较小时（比如20, 10），快排反而比较慢，而插入和希尔排序相对较快，这是因为**插入排序和希尔排序都属于插入类型的排序，而快排和冒泡属于交换类排序**，数据量少时交换所消耗的资源占比大。

数据量较大时，快排果然还是名副其实的快：当数据集达到十万级别时，冒泡排序已经用时800多秒，而快排只用了0.3秒，相信随着数据量的增大，它们之间的差距也会越来越大。

在基本有序数据排序结果中，当n=10和n=100中都是插入排序消耗时间更短，因为数据基本有序，所以需要插入的次数比较少，尽管插入排序需要一个一个比较，但因为数据量不大，所以比较所消耗的资源占比不会太大。

所以数据量大时可先使用 Quick Sort，等基本有序，并且数据量比较小时再使用 Insert Sort（对少量的基本有序数据集比较好）。我个人的测试认为数据集大小为 **30左右** 的时候，Insert 和 Merge 差不多。

6 树

6.3 术语和定义

节点

节点是树的基础部分。它可以有自己的名字，我们称作“键”。节点也可以带有附加信息，我们称作“有效载荷”。有效载荷信息对于很多树算法来说不是重点，但它常常在使用树的应用中很重要。

边

边是树的另一个基础部分。两个节点通过一条边相连，表示它们之间存在关系。除了根节点以外，其他每个节点都仅有一条入边，出边则可能有多条。

根节点

根节点是树中唯一没有入边的节点。在图 6-2 中，/就是根节点。

路径

路径是由边连接的有序节点列表。比如，哺乳纲→食肉目→猫科→猫属→家猫就是一条路径。

子节点

一个节点通过出边与子节点相连。在图 6-2 中，log/、spool/和 yp/都是 var/的子节点。

父节点

一个节点是其所有子节点的父节点。在图 6-2 中，var/是 log/、spool/和 yp/的父节点。

兄弟节点

具有同一父节点的节点互称为兄弟节点。文件系统树中的 etc/和 usr/就是兄弟节点。

子树

一个父节点及其所有后代的节点和边构成一棵子树。

叶子节点

叶子节点没有子节点。比如，图 6-1 中的人和黑猩猩都是叶子节点。

层数

节点 n 的层数是从根节点到 n 的唯一路径长度。在图 6-1 中，猫属的层数是 5。由定义可知，根节点的层数是 0。

高度

树的高度是其中节点层数的最大值。图 6-2 中的树高度为 2。

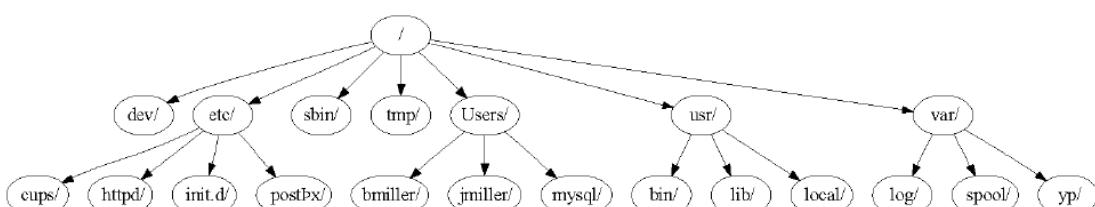


图 6-2 Unix 文件系统的一小部分

- 边：两个节点的连线；
- 节点也叫key；
- 可用Root、Child、branch表示；

6.4 实现方法

6.4.1 list of list

不怎么用

- 基本结构: `[root value, [left branch], [right branch]]`
- 叶子: `[value, [], []]`
- 所以每一个节点要么是长度为3, 要么是`[]`;
- 这个不仅可以二叉树, 其他树也可以;

根据 6.3 节给出的定义, 可以使用以下函数创建并操作二叉树。

- `BinaryTree()` 创建一个二叉树实例。
 - `getLeftChild()` 返回当前节点的左子节点所对应的二叉树。
 - `getRightChild()` 返回当前节点的右子节点所对应的二叉树。
 - `setRootVal(val)` 在当前节点中存储参数 `val` 中的对象。
 - `getRootVal()` 返回当前节点存储的对象。
 - `insertLeft(val)` 新建一棵二叉树, 并将其作为当前节点的左子节点。
 - `insertRight(val)` 新建一棵二叉树, 并将其作为当前节点的右子节点。
- `insert`实际上只插入了一个节点;
 - 假如 `insert val` 时, 原来的位置有棵树 `oldTree`, 那 `insertLeft`会把这个位置变成 `[val, oldTree, []]`;
 -

6.4.2 节点与引用

遵循面向对象, 常用

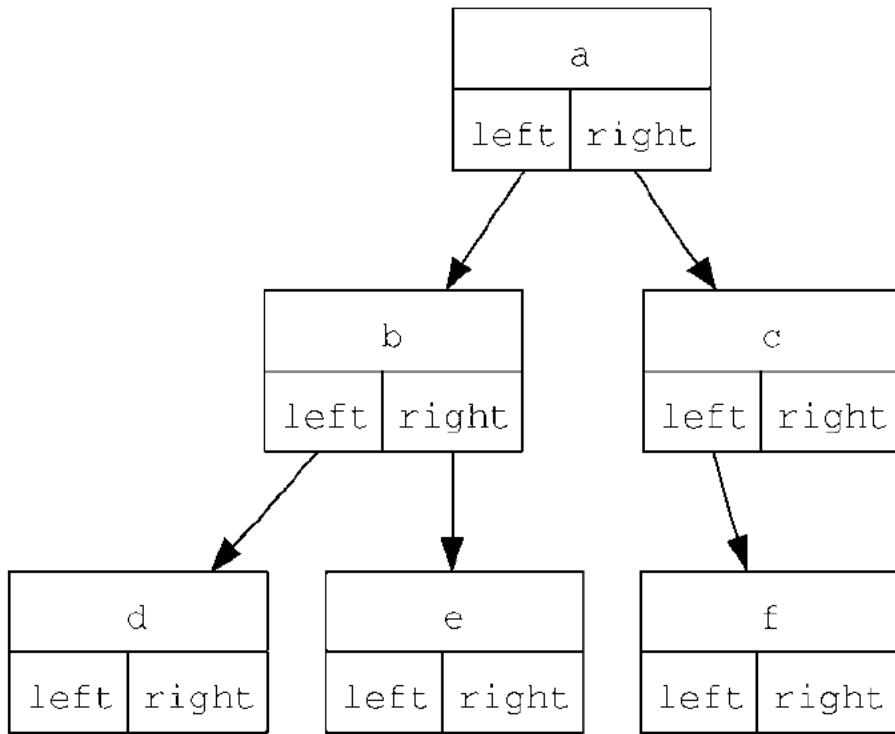


图 6-8 “节点与引用”表示法的简单示例

二叉树的每个子树都是二叉树。

6.5 二叉树应用

6.5.1 解析树

解析树：是语法分析结果的一种表现形式，通常以树状表示语言的语法结构。

```
((7 + 3) * (5 - 2)) ->
```

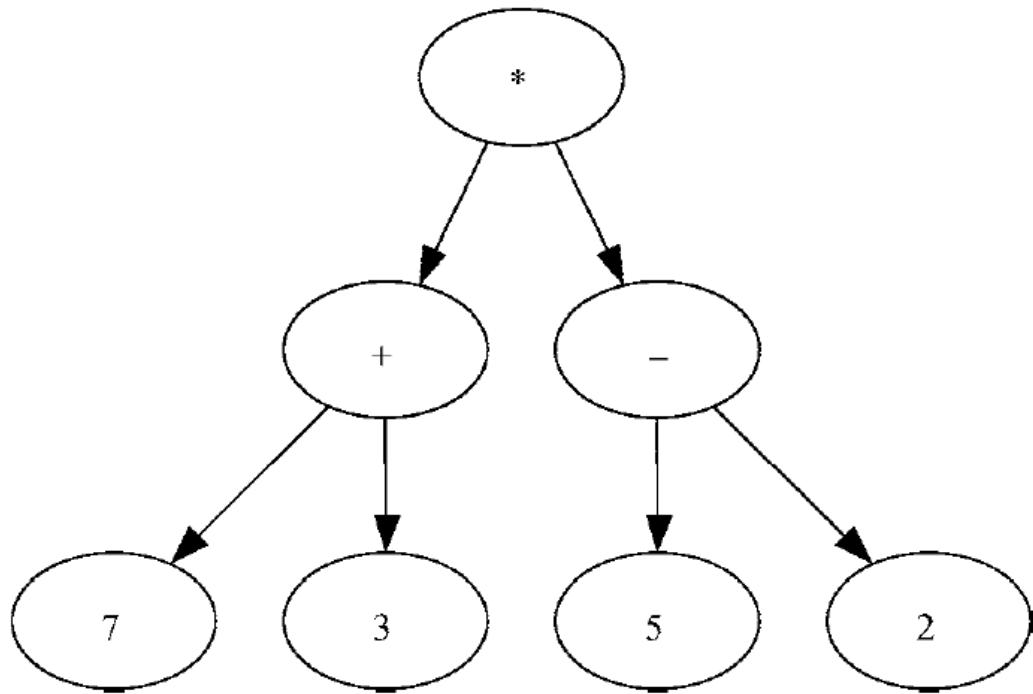


图 6-10 $((7 + 3) * (5 - 2))$ 的解析树

从一棵空树开始：

- 左括号：为当前节点加一个左子节点，并且下沉到该节点；
- 数字：当前节点key，并返回父节点；
- 操作符：当前节点key，添加右子节点，并且下沉到该节点；
- 右括号：返回父节点；

只能是正整数的加减乘除

在这里，所有的父节点满足FILO，可以用栈回退节点。

以 $(1+2)$ 为例，push两次，pop三次，所以一开始应该把根节点，或者空节点push进去。

操作符如何使用：

- import operator
- dict: '+' : operator.add

6.5.2 树的遍历

三种遍历：

- 前序遍历：先访问根节点，然后递归地前序遍历左子树，最后递归地前序遍历右子树。
 $(*+73-52)$
- 后序遍历：先递归地后序遍历右子树，然后递归地后序遍历左子树，最后访问根节点。
 $(25-37+*)$
- 中序遍历：先递归地中序遍历左子树，然后访问根节点，最后递归地中序遍历右子树。
 $(7+3*5-2)$ ，可以还原原来的表达式

也可以改成非递归，以中序为例：

1. 找到最小节点，并把路径上的节点都 push；

2. pop, print

1. 如果存在右子树，把从右子节点，到右子树的最小节点的全都push；
2. 如果不存在右子树，pass
3. 循环
- 3.

6.6 二叉堆实现优先队列

Binary Heap

每个父节点都小于等于左右节点，但是并不意味着左右节点有大小关系

优先队列：和队列一样FIFO，但是队列中的元素是有优先级的，按照优先级IO。I的时候就要根据特征进行优先级排列。

可以用排序 $O(n \log n)$ 和插入 $O(n)$ ，但是更快的方法是**二叉堆——可实现 $O(\log n)$** 。分为最小堆（最小的元素在队首）和最大堆（最大的元素在队首）。

6.6.2 基本方法及其实现

可以实现一个 $O(n \log n)$ 的排序算法

- BinaryHeap()新建一个空的二叉堆。
- insert(k)往堆中加入一个新元素。

- findMin()返回最小的元素，元素留在堆中。
- delMin()返回最小的元素，并将该元素从堆中移除。
- isEmpty()在堆为空时返回 True，否则返回 False。
- size()返回堆中元素的个数。
- buildHeap(list)根据一个列表创建堆。

为了保持 $O(\log n)$ 这样良好的性能，应该让左右子树大小差不多，而不是一边特别大一边特别小——维持树的平衡。

- 满二叉树：每一层都是满的；
- 完全二叉树（此处使用）：除了最后一层，其他都是满的；并且最后一层从左往右排列；

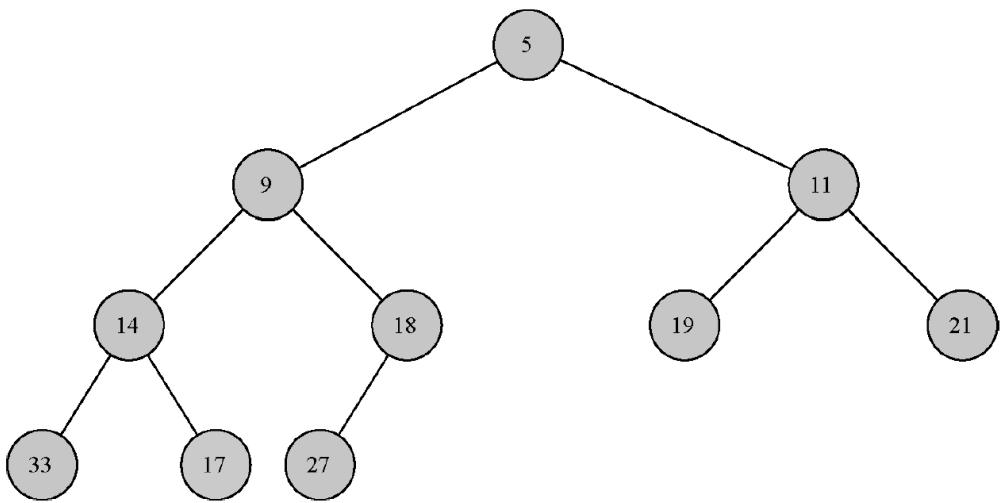


图 6-14 完全二叉树

完全二叉树的神奇性质：

- 不需要list of list / node and reference，可以用一个列表表示，因为——
- 对于深度为k的一层，有 2^{k-2} 个左节点，左节点都是偶数；右节点都是奇数；
- 其中左节点分别为： $2^{k-1}, 2^{k-1} + 2 \times 1, \dots, 2^{k-1} + 2 \times (2^{k-2} - 1) = 2^k - 1$ ；
- 深度为k-1的一层，一共有 2^{k-2} 个节点，分别为
 $2^{k-2}, 2^{k-2} + 1, \dots, 2^{k-2} + 2^{k-2} - 1 = 2^{k-1} - 1$ ；
- 不难发现这两层是2倍的映射关系；
- 所以如果一个列表从1开始index（为了实现这点，可以让第堆顶元素为一个最小值，如果要添加的元素都大于0的话），`[p]`的左节点为`[2p]`，右节点为`[2p+1]`；
- `[p]`的父节点为`[p//2]`；
- 以上的神奇性质使得二叉堆可以用一个列表表示；

堆的有序性：每个父节点都小于等于左右节点，但是并不意味着k层完全大于k-1层，比如上面那张图；

具体的实现：

insert $O(\log n)$

为了保证是个完全二叉树，应该加到队伍末尾，但是可能会破坏堆的有序性，因此需要将其进行交换。假设一个父节点只有一个子节点（子节点 > 父节点），又加入了一个节点，那么可能会有新节点 < 父节点 < 子节点，因此应该让新节点当父节点：

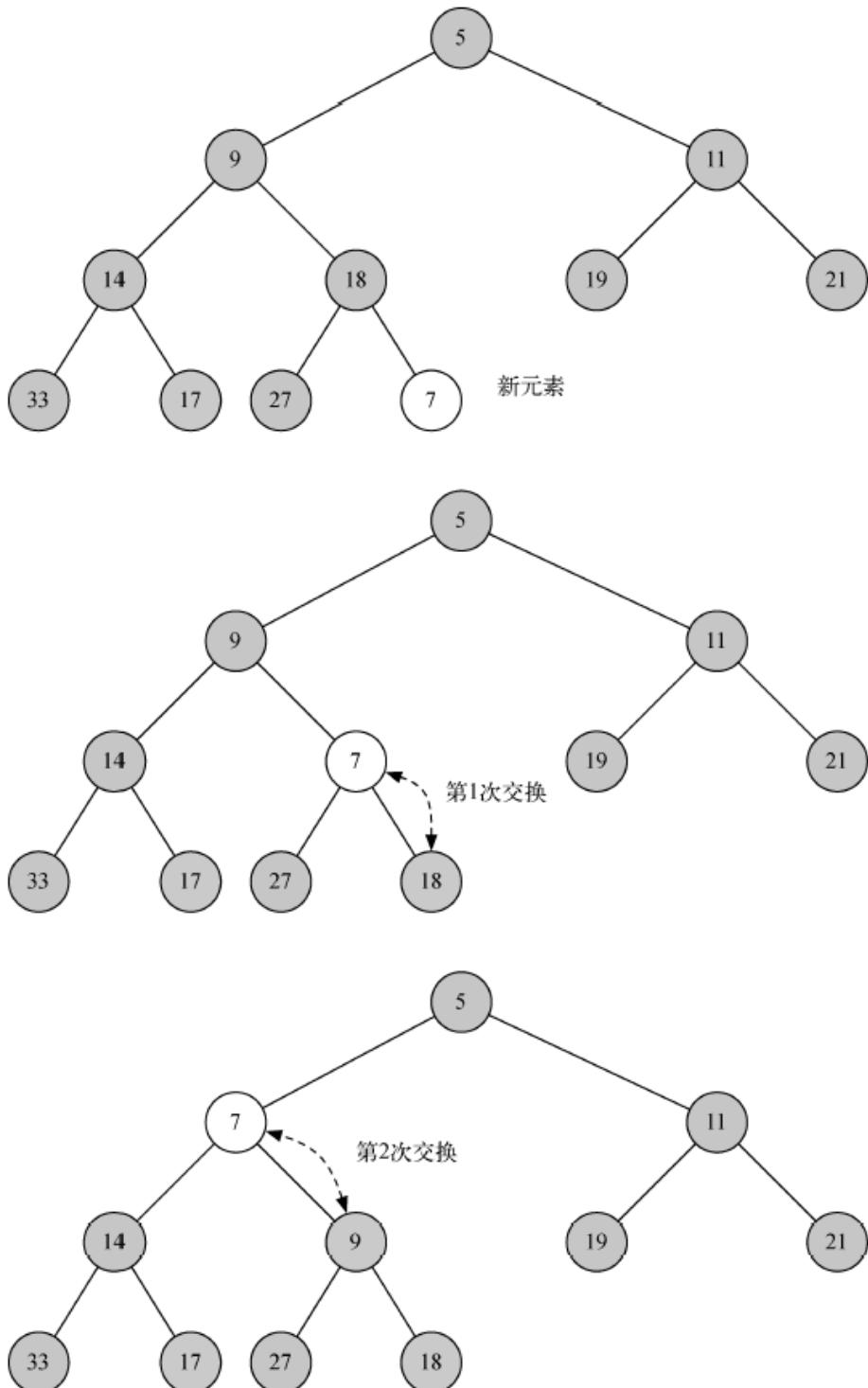


图 6-16 将新元素往上移到正确位置

可以用递归或者循环，感觉似乎没差别，但是没有验证；

delMin $O(\log n)$

找到最小的很简单，难点在于如何重建。

为了维持完全二叉树的形态，应该把列表最后一个移动到堆顶；然后根据parent和child的关系不断移动这个元素。

- 如果parent小于两个childs，那就没问题；
- parent应该和两个child中最小的交换；
- 循环：直到parent小于两个childs，终止；

- **交换**: 在这个过程中, 交换后要先确认一个childMin, 再让childMin和parent交换。然后parent为childMin, 两个child根据parent可计算出;
- **问题**:
 - 但如果parent在最后一层, child就会越界, 就没法比较
 - parent为末尾节点的父节点, 并且该父节点只有一个子节点, 也就是single child;
 - 可以用一个minChild函数, 判断存在几个child, 并且返回相应的值;
- 如果堆里只有1个元素 (不包括为了凑数的那个), 应该直接执行, 不然把最后一个元素放到堆顶的操作会有问题;

buildHeap $O(n)$

- 一种想法是一个一个insert ($O(n \log n)$) ;
- 一种想法是既然这是个有序数组, 可以用二分法 ($O(\log n)$) 找到插入的位置, 但是可能会导致其他元素都移动 ($O(n)$) —— 这样总体为 ($O(n \log n)$) ;
- 书上说可以做到 $O(n)$, 但是证明太复杂不考虑;
- 我想可以用py自带的排序, 底层优化还是快的, 直接extend;

书上的方法:

1. 直接extend;
2. 所有父节点的index是从1到 `len(new_list) // 2`, 只要考虑移动这些父节点就行;
3. 从最后一个父节点开始往前遍历;
4. 如果父节点大于子节点, 就向下一直移动, 直到满足有序性;

| 这里向下交换以满足堆的有序性的过程和delMin一样, 可以把switchDown抽出来;

| 相比switch down, 常用 perc down: percolate(过滤)

6.7 二叉搜索树

和二叉堆不一样, 二叉搜索树的目的是为了实现二分查找, 因此其各节点之间的关系和之前不一样: `childLeft < parent < childRight`。

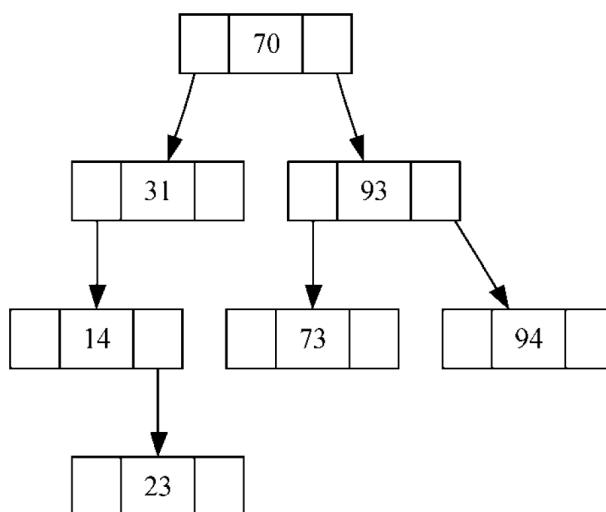


图 6-20 简单的二叉搜索树

基本代码

书中直接给出

在这里的 `TreeNode` 更加复杂，特点包括：

- `payload`: 即键值对中的值；
- 显式储存的 `parent`；
- 方法：
 - 是否有LR子节点，有的话返回；
 - `isChild`
 - `isRoot`
 - `isLeaf`
 - 几个 `children`
 - 更换数据**但是不更换parent**

外部的 `BinarySearchTree`:

- 包含一个迭代器 `__iter__` ?

put——生成或插入节点

`put, __put, __setitem__`

`put`: 检查有没有根节点，没有的话就 `TreeNode -> root`；

有的话，就要按其他方法搜索位置。私有的辅助函数 `_put(self)` :

1. 从根节点开始，比较键，然后决定左右；
2. 如果左或右节点节点为空，说明有地方可以放；没地方的话就递归；
3. `new TreeNode()`, `insert`;
4. 如果key一样，就更新value

get——获取value

`get, __get, __getitem__, __contains__`

和之前的差不多。

注意：

- `get` 获取值
- `__get` 获取键所在的Node
- `__getitem__` 重构 `[]`，可通过 `[k]` 访问v
- `__contains__` 重构 `in`

delete——删除节点

`delete()`, `__delitem__()`, `remove()`

delete主函数

基本思路：

1. `_get` 找到待删除的节点；
2. 找不到key就raise error；
3. 找到了就改变size，
 1. 如果这个节点是root, 置None；
 2. `remove`函数

`__delitem__`

简单地调用`delete`

remove

三种情况，取决于这个node有几个children

case 1: no child

删除，并且让parent没有孩子

但是要分清LR，以此判断删除parent的那个attr

case2: single child

让孩子替代待删除节点。

但是要分情况：待删除节点LR，因为要改变parent的LR；

孩子改parent就行

case3: both children

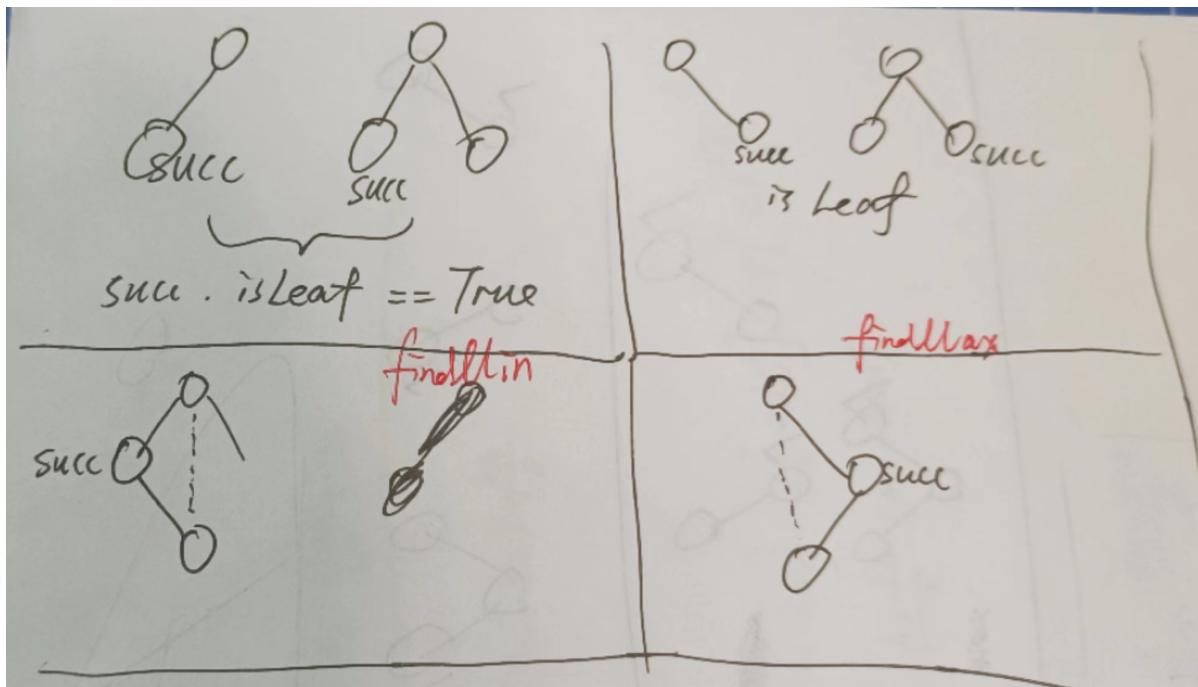
不太可能用其中一个子节点代替，所以要想别的方法。

能不能找到一个**后继元素**代替这个节点，同时保持二叉搜索树的特征呢？

可以用**左子树的最大节点**，或者**右子树的最小节点**来代替待删除节点。

这里可以分成三个部分：

1. `findMin`：对于一个TreeNode，返回以这个节点为根节点的子树中的最小键，也就是一直往左走；
2. `findSuccessor`：对于一个待删除的节点，找到右子树的最小键，作为 succ 后继节点返回；
(这个函数里有别的condition分支，还有其他用途)
3. `spliceout`：删除 succ 并重建联系，这里如果递归调用 `delete` 会慢。该函数适合 `findMin` 和 `findMax`，因为 succ 的位置是比较明确的（对于删除操作）。



6.8 平衡搜索树 AVL

6.8.1 概述

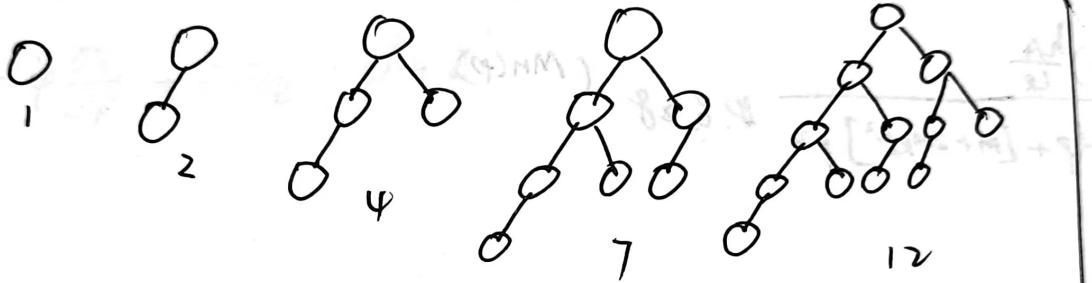
构建BST的时候，有可能构建出一个全是左子节点的树，相比于左右完全平衡的树，各项操作会从 $O(\log n)$ 变成 $O(n)$ ，因为时间复杂度取决于树的深度。

AVL树是一种自动保持左右平衡的树，通过记录每个节点的平衡因子（balance factor）来保持平衡。

和红黑树不一样，据说红黑树更好。

平衡因子：左右子树高度之差：

- 为-1、0和1的时候都可以算平衡；
- 如果大于0就是左倾，左边更深；
- 小于0是右倾，右边更深；



$$4 = 1 + (1+2)$$

$$7 = 1 + (2+4)$$

$$12 = 1 + (4+7)$$

$$N_h = 1 + N_{h-1} + N_{h-2}$$

这个近似于斐波那契数列，而斐波那契数列两项之比趋近于 $\frac{1+\sqrt{5}}{2}$...略去数学过程，可得：
 $h = 1.44 \log_2 N_h$ 。

6.8.2 AVL实现

插入 put set

在NodeTree新增属性 balanceFactor，新class继承自 BinarySearchTree。

平衡因子：左边高度减去右边高度。

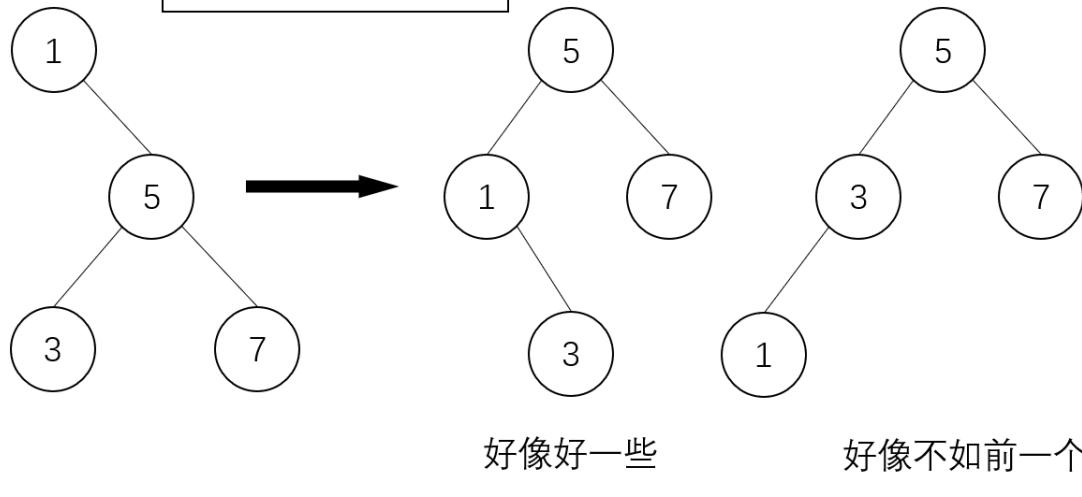
put: overload，需要在插入的同时更新所有父节点的平衡因子。向上更新的过程中：

1. 如果一个平衡因子被变成了0，那就不会再影响上面的父节点；
2. 到了根节点，停止；
3. 如果一个节点的平衡因子不是-1、0或者1，就进行再平衡；

再平衡 rebalance

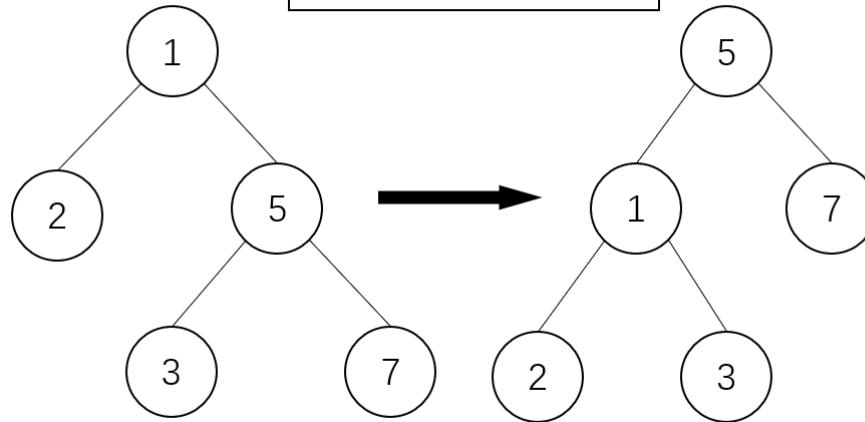
左旋实现

右倾 -> 左旋



好像不如前一个

右倾 -> 左旋

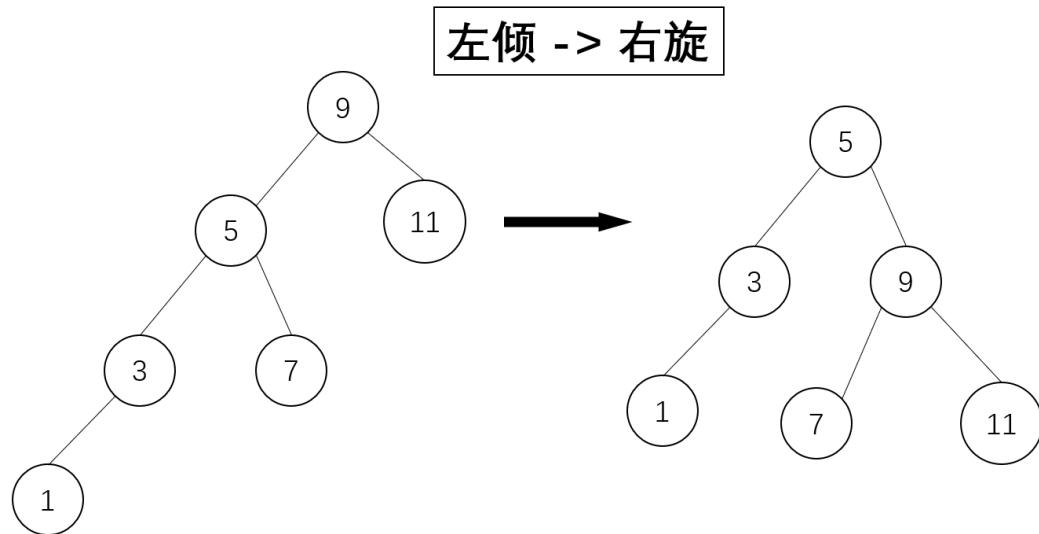


这个例子更能体现优势

具体的重建connection可分为三部分：

1. `newRoot`, 和 `oldRoot.parent` 的连接;
2. 把节点 3 和 `oldRoot` 连接;
3. 把 `oldRoot` 和 `newRoot` 重建连接;
4. 重新计算父节点的平衡因子：
 1. 旋转前后, 非父节点的高度是不变的, 因此首先把两个父节点的平衡因子用非父节点的高度表示;
 2. 写出旋转后父节点的平衡因子 (用同样的非父节点的高度表示) ;
 3. 对于每个节点, 计算其前后变化;
 4. 能用到的数学: $\max(x, y) = -\min(-x, -y)$,
 $\max(x, y) - a = \max(x - a, y - a)$;

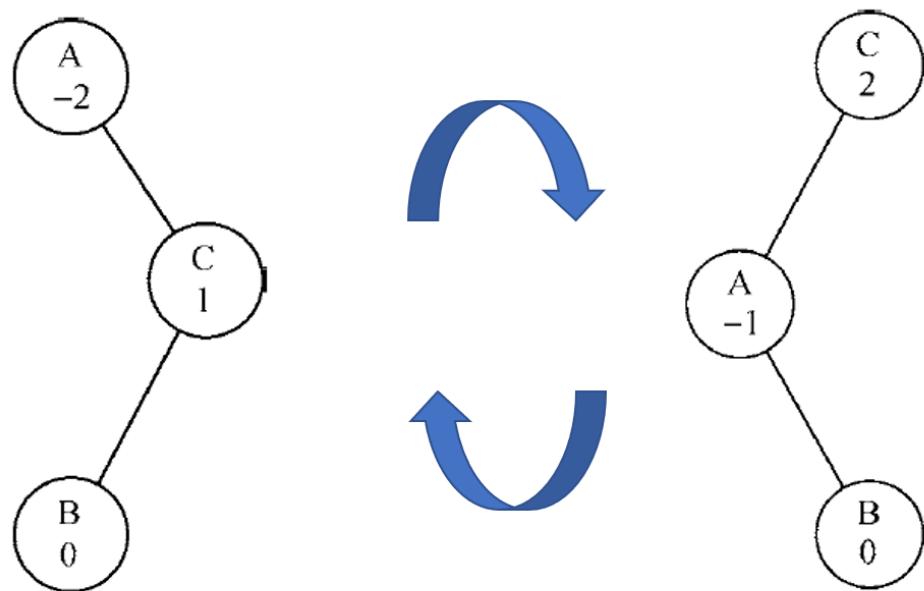
右旋实现



类似左旋

旋转的时机

一个怎么左旋右旋都不行的例子：



要解决这种问题，必须遵循以下规则：

- 如果子树需要左旋，首先检查右子树的平衡因子。如果右子树左倾，就对右子树做一次右旋，再围绕原节点做一次左旋；
- 如果子树需要右旋，首先检查左子树的平衡因子。如果左子树右倾，就对左子树做一次左旋，再围绕原节点做一次右旋；
- **如果右倾，就让右边的更加右倾；如果左倾，就让左边的更加左倾；**

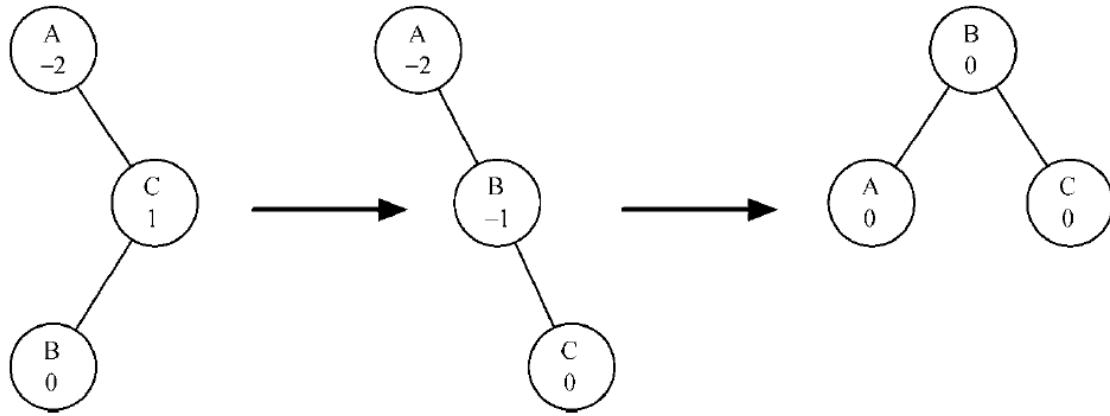


图 6-33 先右旋，再左旋

delete

1. 如果是 leaf，那就没事：更新向上的父节点 balance factor，必要的时候 rebalance；
2. 如果不是leaf：找 succ 替代，并重建 succ 附近的连接 (`TreeNode.spliceout`)；
3. 从 succ 的父节点开始向上更新；

7 图(Graph)

7.2 术语和定义

有一些概念和树类似：

- 顶点 V, **Vertex is singular of Vertices** (包括 key, payload)；
- 边 E, Edge：可以是单向、双向的；如果所有边都是单向的，称之为有向图；
- 权重：边可以带有权重——边可用三元组表示：`(start, end, weight)`；
- 子图：由一部分边和顶点构成的；
- 路径：由边连成的；
- 环：其中**有向无环图(DAG, Directed Acyclic Graph)**很有用；

图 7-2 展示了一个简单的带权有向图。我们可以用 6 个顶点和 9 条边的两个集合来正式地描述这个图：

$$V = \{V0, V1, V2, V3, V4, V5\}$$

$$E = \left\{ \begin{array}{l} (v0, v1, 5), (v1, v2, 4), (v2, v3, 9), (v3, v4, 7), (v4, v0, 1), \\ (v0, v5, 2), (v5, v4, 8), (v3, v5, 3), (v5, v2, 1) \end{array} \right\}$$

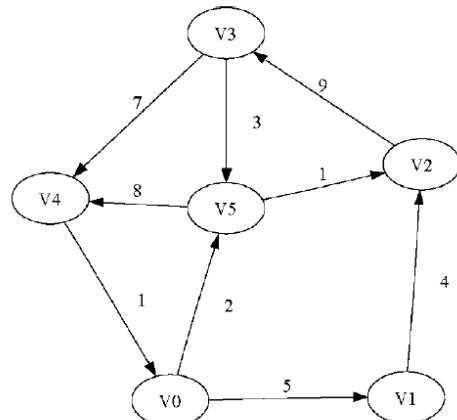


图 7-2 简单的带权有向图

7.3 抽象数据类型及其实现

图的抽象数据类型由下列方法定义。

- `Graph()` 新建一个空图。
- `addVertex(vert)` 向图中添加一个顶点实例。
- `addEdge(fromVert, toVert)` 向图中添加一条有向边，用于连接顶点 `fromVert` 和 `toVert`。
- `addEdge(fromVert, toVert, weight)` 向图中添加一条带权重 `weight` 的有向边，用于连接顶点 `fromVert` 和 `toVert`。
- `getVertex(vertKey)` 在图中找到名为 `vertKey` 的顶点。
- `getVertices()` 以列表形式返回图中所有顶点。
- `in` 通过 `vertex in graph` 这样的语句，在顶点存在时返回 `True`，否则返回 `False`。

两种实现方式：邻接矩阵和邻接表

7.3.1 邻接矩阵

矩阵的行和列代表顶点；矩阵内的元素代表是否连接，以及权重；

但由于大部分情况下顶点之间的连接是比较稀疏的，因此在 python 里实现一个二维矩阵消耗资源较大；

而且在 python 里实现动态的增加、删除节点应该也不容易；

	V0	V1	V2	V3	V4	V5
V0		5				2
V1			4			
V2				9		
V3					7	3
V4	1					
V5			1		8	

图 7-3 邻接矩阵示例

7.3.2 邻接表

为了实现稀疏连接的图，更高效的方式是使用邻接表。

在邻接表实现中，我们为图对象的所有顶点保存一个主列表，同时为每一个顶点对象都维护一个列表，其中记录了与它相连的顶点。在对 Vertex 类的实现中，我们使用字典（而不是列表），字典的键是顶点，值是权重。

```

1 class Graph -> list of vertices
2
3 class Vertex -> dict of info: {neighbor vertex: edge weight}

```

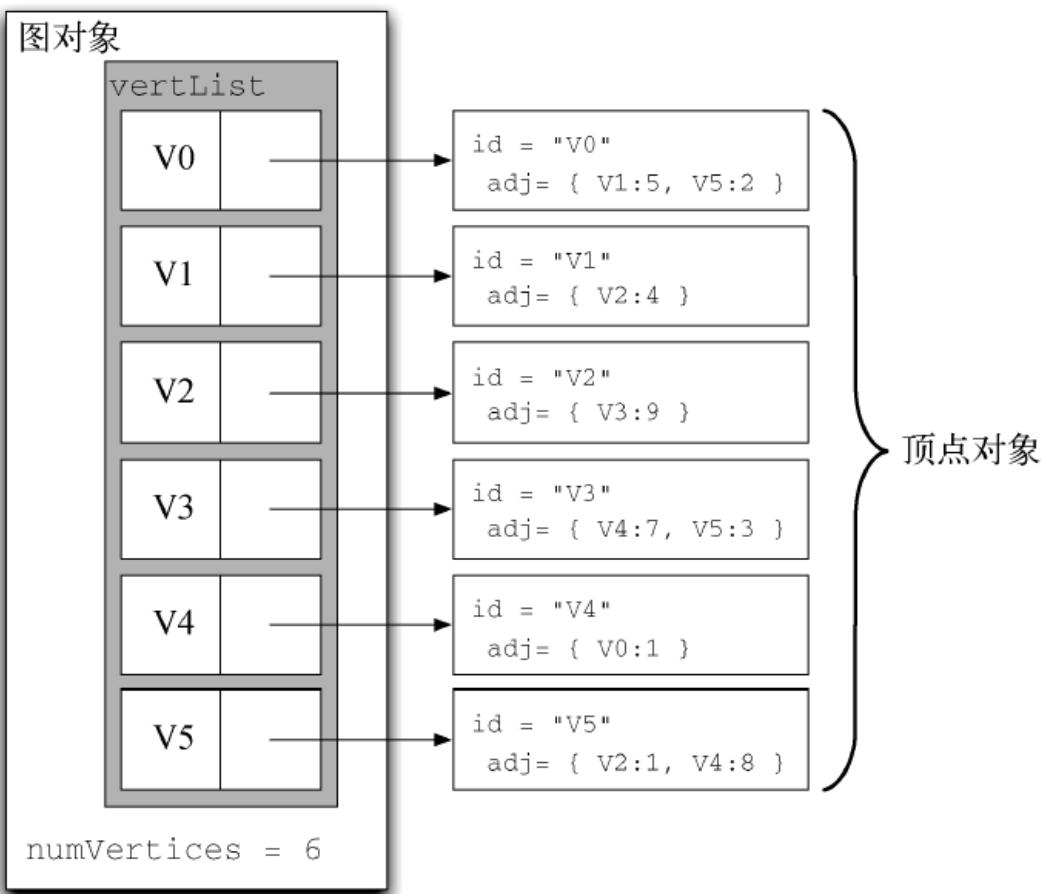


图 7-4 邻接表示例

Vertex类实现

```

1 self.id: usually a string
2 self.connectedTo: a dict of nbr: weight
3
4 methods:
5     addNeighbor
6     __str__, __repr__
7     getConnections: return self.connectedTo.keys()
8     getId: self.id
9     getWeight(nbr): look up dict

```

Graph类实现

```

1 self.vertList = {id: type<'Vertex'>}
2 self.numVertices
3
4 methods:
5     addVertex(key): add a vertex without edge
6     getVertex(key): return self.vertList.get(key)
7     __contains__: reload in
8     __iter__: reload, iter(self.vertList.values())
9     addEdge(from, to, weight): ENSURE from and to in self first
10    getVertices: return dict.keys()
11

```

key 就是 id

7.4 BFS(Breadth First Search): Word Ladder Game

7.4.1 Intro

Word Ladder: 该游戏有一个起始词和一个终止词，游戏者需要发现一条连接两个词的词汇链，**词汇链上的两个相邻词只差一个字母**。游戏者一般是更改起始词中的一个字母，获得一个新词，然后继续更改所得的新词中的某个字母，再获得一个新词，最终获得终止词。

7.4.2 Build Graph

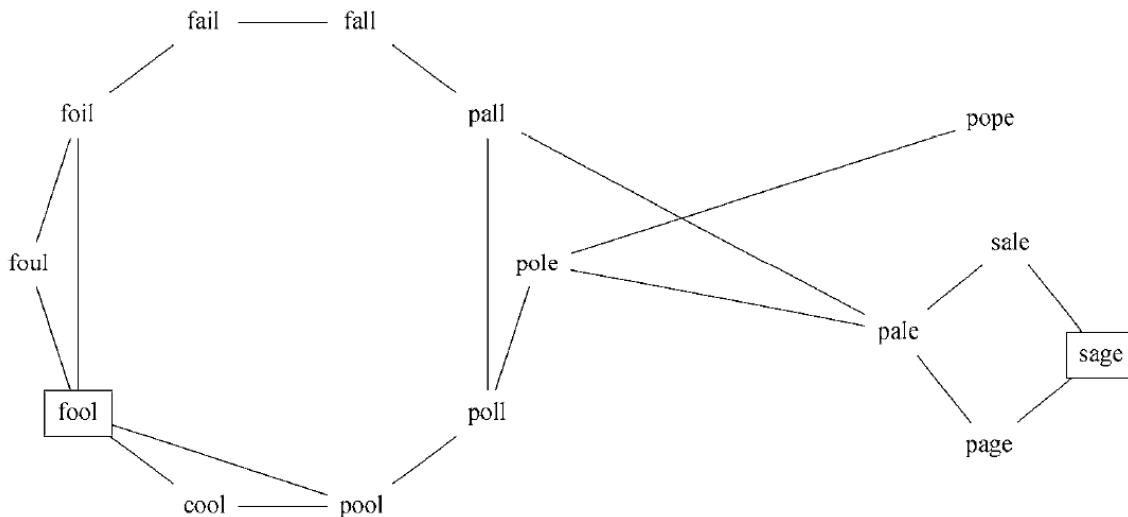


图 7-5 用于解决词梯问题的小型图

A graph without weight and direction

为了构建这个图，需要准备一个单词列表。对于每个单词，和其他单词比较是否只相差一个字母：正确的话就创建一条边。**但这个想法的时间复杂度是 $O(n^2)$** 。

所以：

1. 先筛选和目标单词有关的词库，比如把四个字母的单词都筛选出来；
2. 对于每个单词，比如 shit，分别匹配满足 .hit/s.it/sh.t/sh.. 的单词，把他们分别放进 4 个篮子里；
3. 对于每个篮子里的单词，都可以通过一个字母来转换，所以没个篮子内部应该相互连接；

7.4.3 BFS Realize

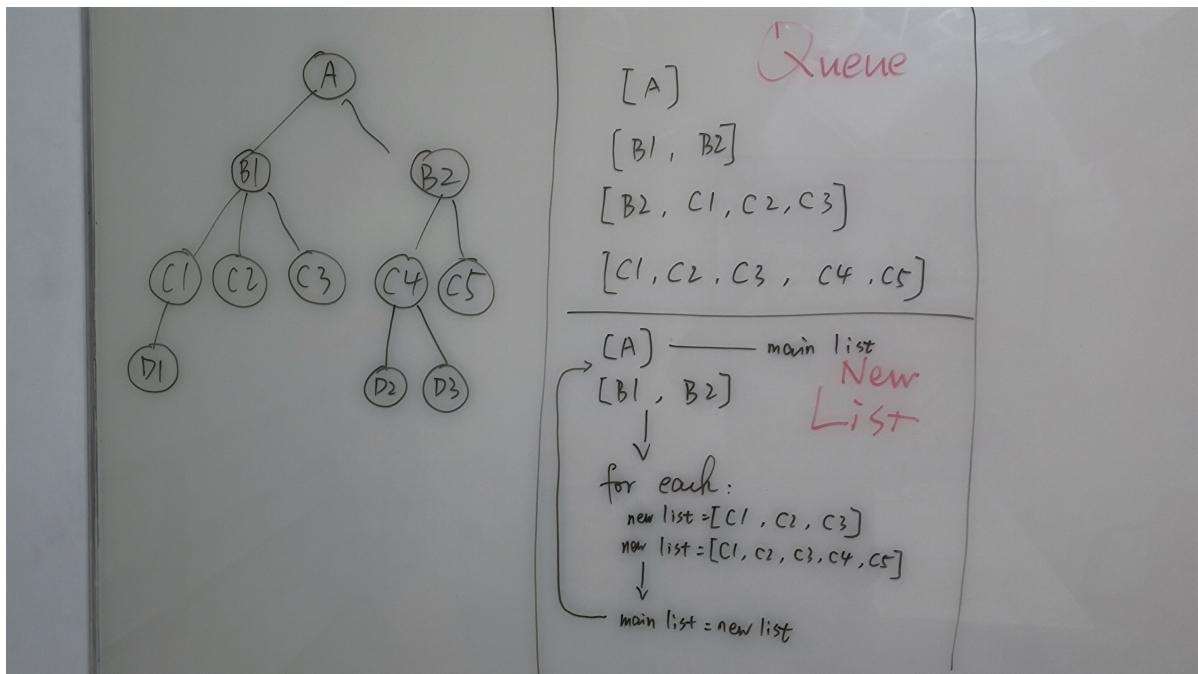
给 vertex 类增加几个属性：

- ```
1 | distance: 第一个 vertex 是0，第一层是1，以此类推
2 | predecessor: 标记自己的上级
3 | accessed: 该节点是否被访问过，每一层只访问未被访问过的
```

如果发现某个节点已经被访问过，说明存在更短的路径，所以不应该再次访问：所以能知道是最短路径的长度。

`predecessor` 的作用是回溯，这样就可以知道具体的路径。

两种实现方法，在 python 里没啥区别：



## 7.5 DFS(Depth First Search): Knight's Tour

### 7.5.1 Intro

[骑士周游问题](#): 指给定一个棋盘，骑士可以走日字，问怎么走才能不重复地遍历全部格子。

### 7.5.3 Realize(Basic Method)

用 Stack 表示待探索的队列。每次递归，将栈顶的元素作为当前元素，并将当前元素标记为“已访问”。

1. 递归退出条件：遍历路径数量等于期望值；
2. 递归缩小条件：对于每一个潜在的下一步，push 并递归；
3. 回溯条件：找到终点，或者没有下一步；
4. 如果对所有子元素，都无法找到终点，则将该节点标记为“未访问”，pop

### 7.5.4 Time Complexity Analysis(Basic Method)

最坏的情况下，需要遍历所有可能的路径才行。

假设棋盘是8864个格子，看成一棵树的话，需要找到深度63的分支；

每个父节点可能有2~8个子节点，所以时间复杂度是 $O(k^n)$ 的： $k$ 是平均每个父节点有多少子节点； $n$ 是层数。指数阶的算法，非常耗时。

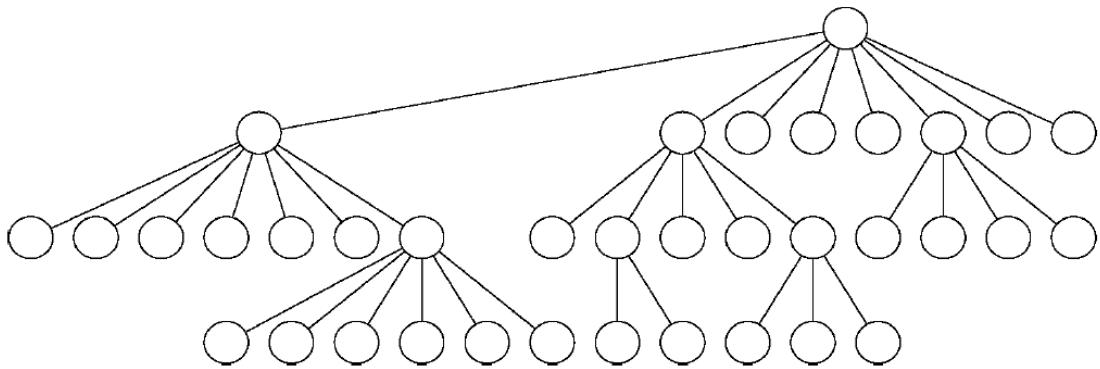


图 7-14 骑士周游问题的搜索树

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 4 | 4 | 4 | 3 | 2 |
| 3 | 4 | 6 | 6 | 6 | 6 | 4 | 3 |
| 4 | 6 | 8 | 8 | 8 | 8 | 6 | 4 |
| 4 | 6 | 8 | 8 | 8 | 8 | 6 | 4 |
| 4 | 6 | 8 | 8 | 8 | 8 | 6 | 4 |
| 4 | 6 | 8 | 8 | 8 | 8 | 6 | 4 |
| 3 | 4 | 6 | 6 | 6 | 6 | 4 | 3 |
| 2 | 3 | 4 | 4 | 4 | 4 | 3 | 2 |

图 7-15 每个格子对应的合理走法数目

## 7.5.tmp1 启发式算法技术

我们称利用已有知识来加速算法为启发式技术。

Warnsdorff 算法：

从上图可以发现，边缘格子的潜在子节点更少。如果我们将当前节点的子节点排个顺序，假如是子节点的潜在走法从多到少，那每次都优先遍历靠近中间的节点，因为中间的走法更多；

为了保证骑士能尽快到达边边角角，对当前节点的子节点按照可能的走法数量从小到大排序。

快了好多！

### 7.5.5 Realize(General Method)

之前的骑士周游问题，最终创建了一棵没有分支的深度优先搜索树，是一种特殊情况；通用的深度优先搜索更简单。

一次深度优先搜索甚至能够创建多棵深度优先搜索树，我们称之为**深度优先森林**。

和宽度优先搜索类似，深度优先搜索也利用**前驱连接**来构建树。

在 `vertex` 中增加两个属性：第一次探索到该点的时间，和结束探索该点的时间。

创建一个新类：`DFSGraph(Graph)`。

```
1 class DFSGraph(Graph):
2 def __init__(self):
3 super().__init__()
4 self.time = 0
5
6 def dfs(self):
7 for vtx in self:
8 if not vtx.accessed:
9 self.dfsvisit(vtx)
10 # 为了不遗漏节点，创建了多棵树；
11 # 如果某个节点已经存在于某棵树内：
12 # 就不会以其为根节点创建树了
13
14 def dfsvisit(self, stVtx: Vertex):
15 self.time += 1
16 stVtx.accessed = True
17 stVtx.distTime = self.time
18
19 for nextVtx in stVtx.getConnections():
20 if nextVtx.accessed:
21 continue
22 else:
23 next.pred = stVtx
24 self.dfsvisit(nextVtx)
25 self.time += 1
26 stVtx.finTime = self.time
```

关于访问时间的一个示意图：

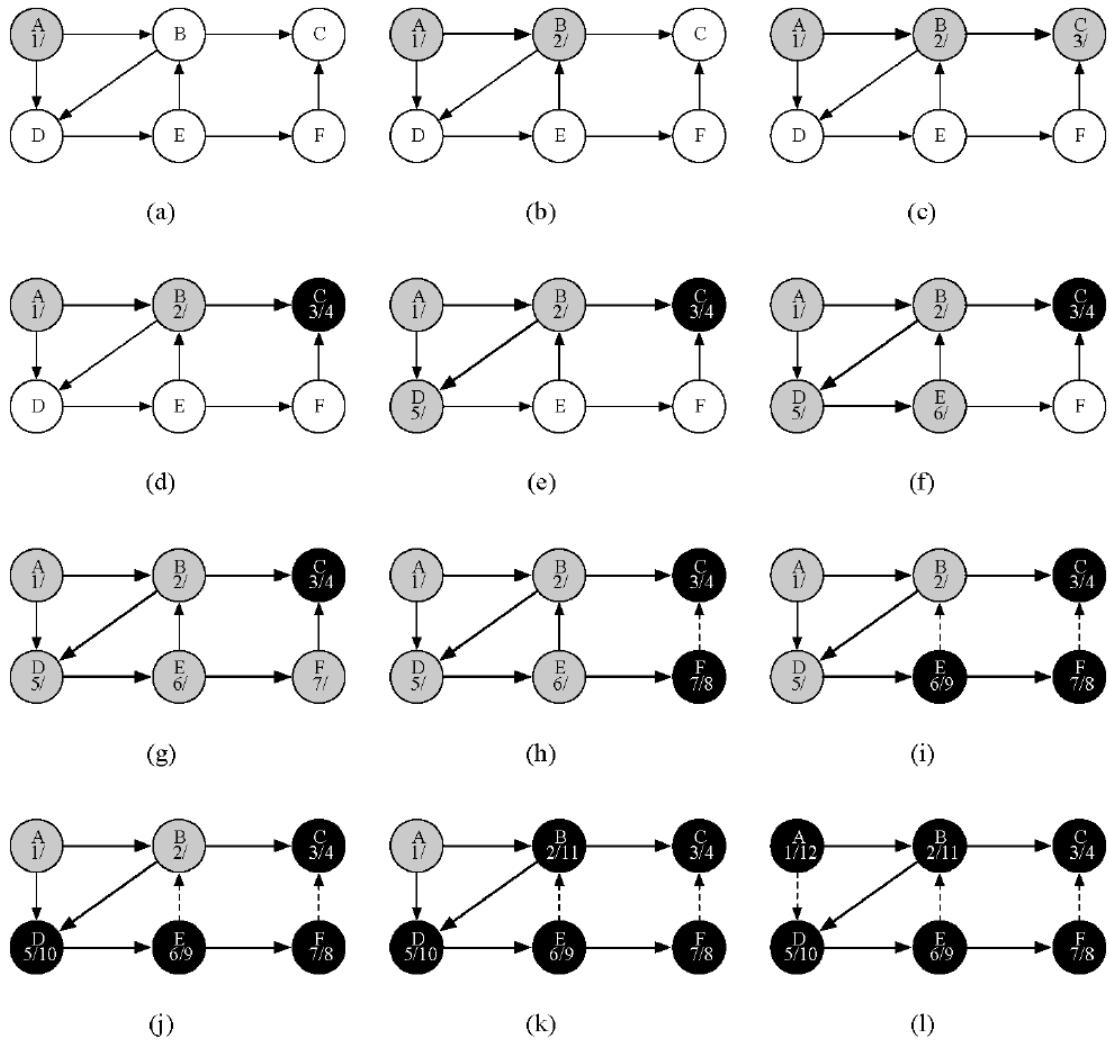


图 7-16 构建深度优先搜索树

**括号特性：**DFS树中的每个子节点的时间都包含在父节点时间之内。

## 7.6 拓扑排序

为了展示计算机科学家可以将几乎所有问题都转换成图问题，让我们来考虑如何制作一批松饼。配方十分简单：一个鸡蛋、一杯松饼粉、一勺油，以及 $\frac{3}{4}$ 杯牛奶。为了制作松饼，需要加热平底锅，并将所有原材料混合后倒入锅中。当出现气泡时，将松饼翻面，继续煎至底部变成金黄色。在享用松饼之前，还会加热一些枫糖浆。

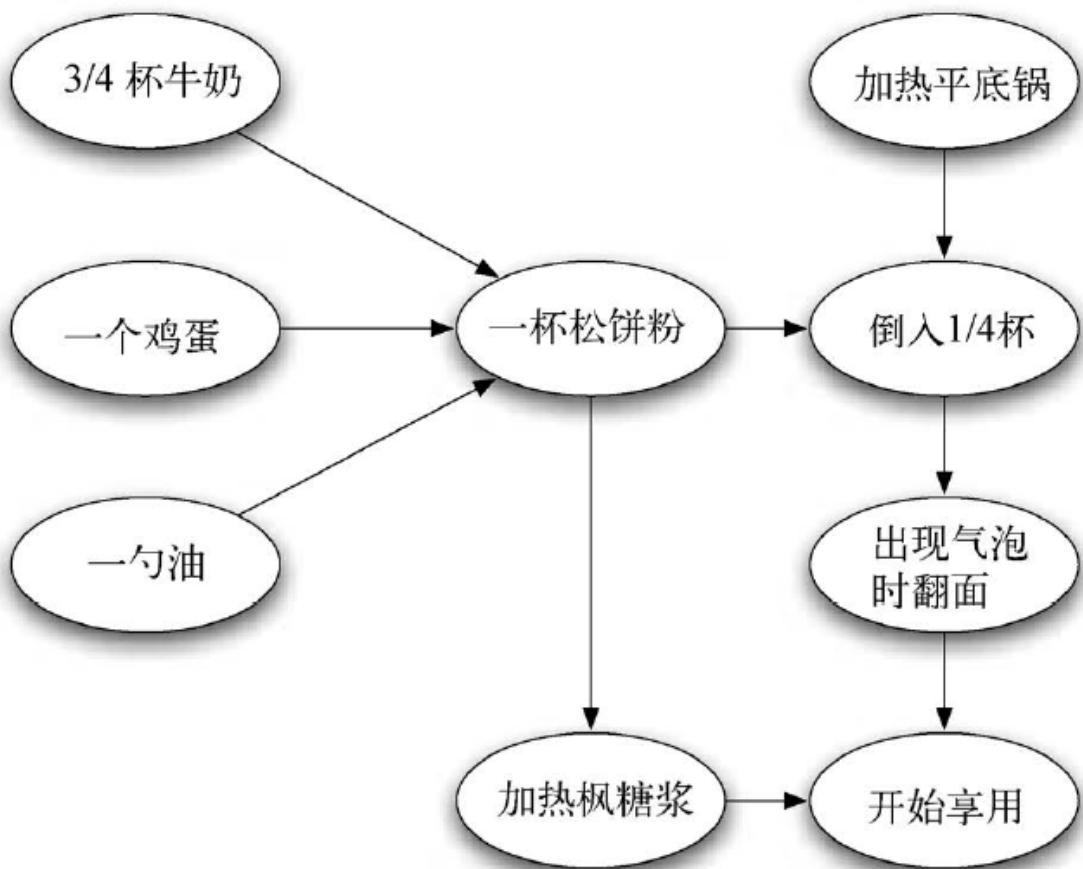


图 7-18 松饼的制作步骤

**拓扑排序**根据有向无环图生成一个包含所有顶点的线性序列，使得如果图G中有一条边为 $v -> w$ ，那么顶点v排在顶点w之前：

因此要想达到w，就得先达到v：意味着应该先做v：

而父节点的结束时间晚于子节点，因此先做结束时间最晚的，最后做的结束时间最早的相当于叶子节点。

在很多应用中，有向无环图被用于表明事件优先级。

制作松饼只是其中一个例子，其他例子还包括软件项目调度、优化数据库查询的优先级表，以及矩阵相乘。

加热平底锅

一勺油

一个鸡蛋

3/4杯牛奶

一杯松饼粉

加热枫糖浆

倒入1/4杯

出现气泡时翻面

开始享用

进程已结束, 退出代码0

根据结束时间从大到小排列

## 7.7 强连通单元(Strongly Connected Algorithm)

强连通单元图算法, 可以找出图中高度连通的顶点簇:

对于图 $G$ , 强连通单元 $C$ 为最大的顶点子集 $C \subset V$ , 其中对于每一对顶点 $v, w \in C$ , 都有一条从 $v$ 到 $w$ 的路径和一条从 $w$ 到 $v$ 的路径。

然后就可以简化图

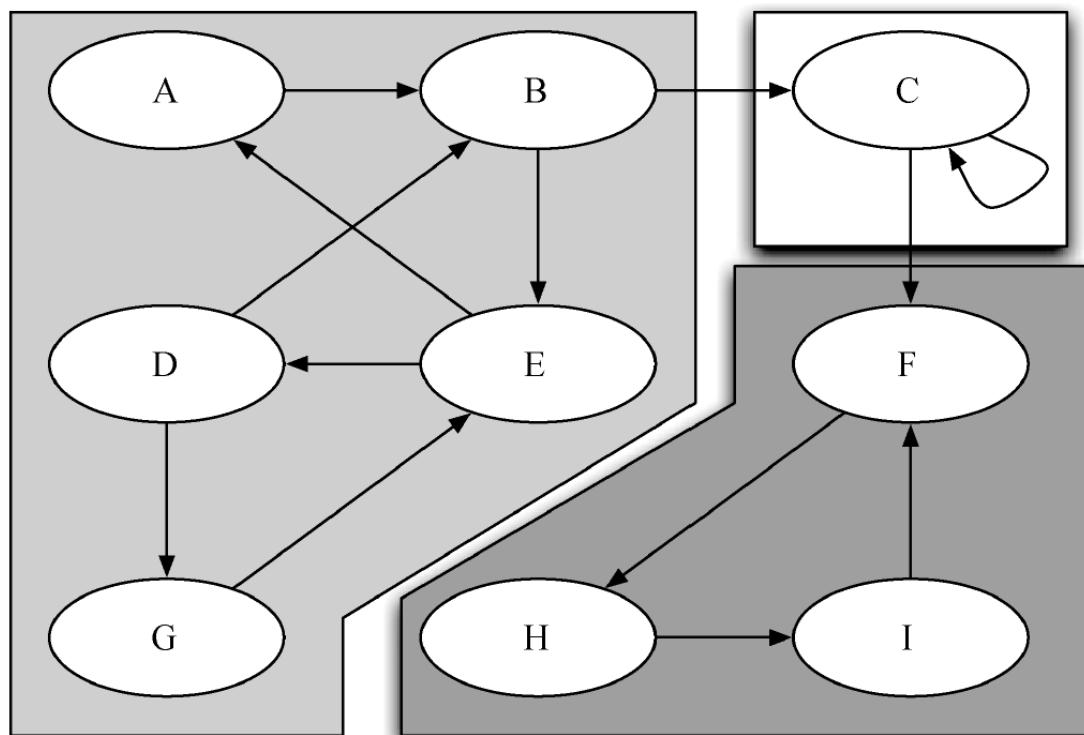


图 7-22 含有 3 个强连通单元的有向图

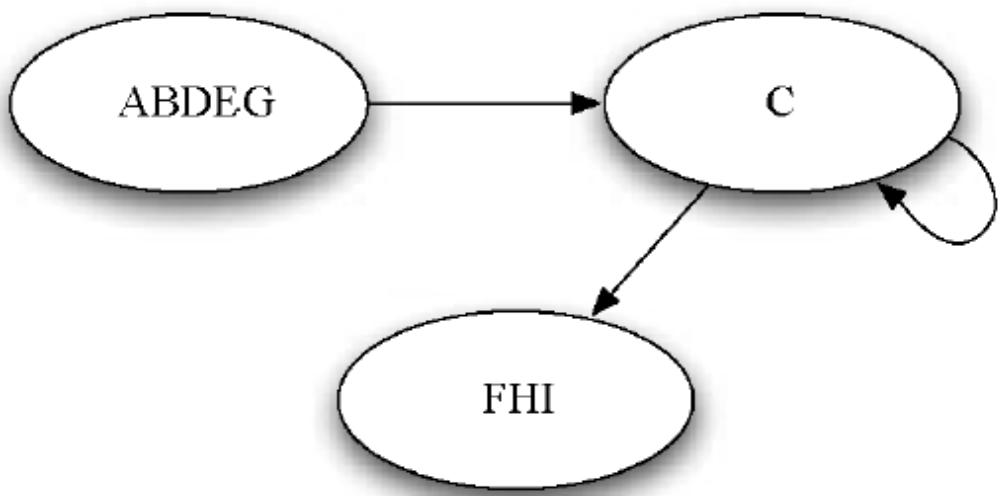
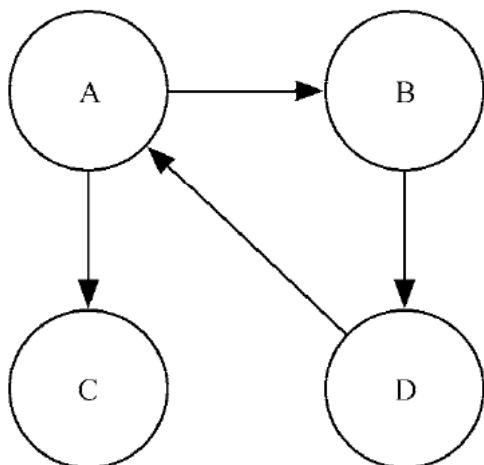


图 7-23 简化后的有向图

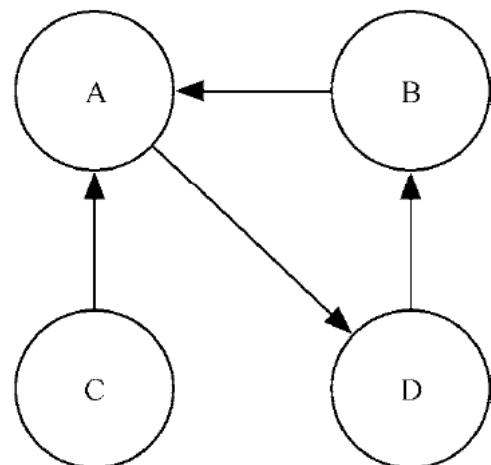
### 7.7.1 强连通算法: Kosaraju

不懂原理

转置图: 把有向图里的边都反过来



(a) 图G



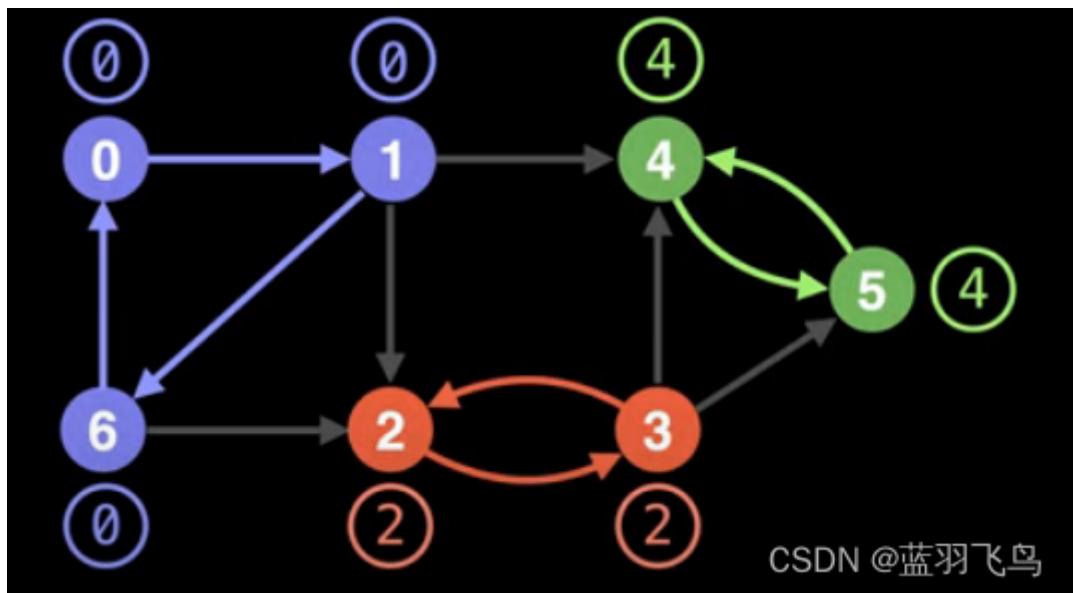
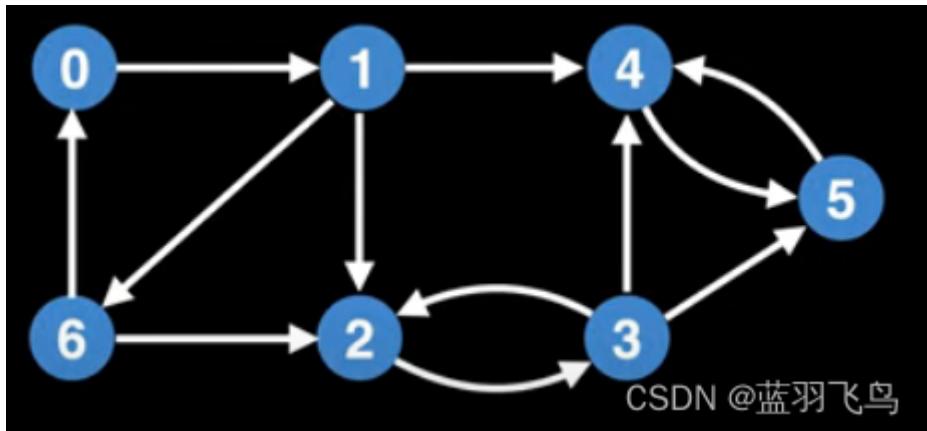
(b) 图G的转置图

图 7-24 图 G 及其转置图

1. 正常建一个`DFSGraph`, 正常`dfs()`后, 把各节点按照**结束访问时间递减排序**;
2. 把上一步得到的各节点`id`, 和反向的`edge`建立一个转置图;
3. 对该转置图正常`dfs()`, 就会按照期望的顺序访问节点;
4. 得到的森林就是SCCs。

## 7.7.2 Tarjan 强连通分量算法: Tarjan's Strongly Connected Components Algorithm

Tarjan算法用于有向图，用于找强连通区域，割边等，属于图论内容。



基本方法：

1. 给每个顶点分配一个id，可以用第一次访问的时间，`dfn(depth first number)`；
2. 每个顶点还有一个`low`值，表示当前node能到达的node中最小的id（包括它自己）。
3. 具有相同`low`值的在同一个SCC；`low == dfn`的是根节点；

---

`dfn`已经有了，问题在于怎么更新`low`：

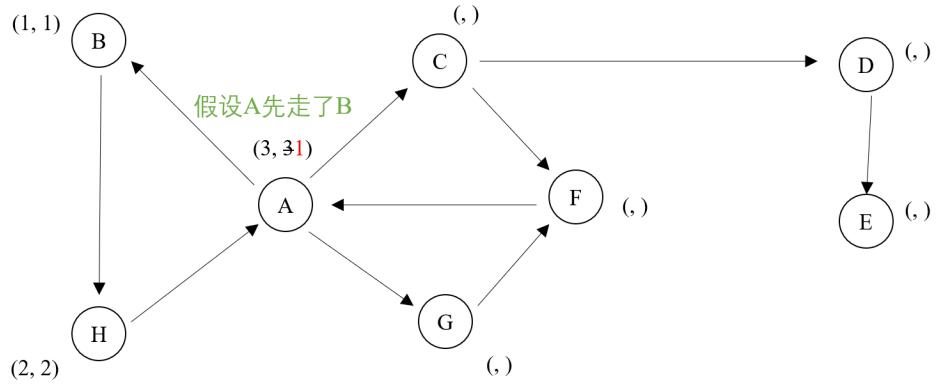
三个点：

1. 不能访问重复节点：这点通过`vertex.accessed`已经实现，和普通DFS差不多；
2. `low`的初值是`dfn`；
3. 对于每个顶点，在遍历所有下家时，实时更新`low`值，其`low`应为本身和下家`low`的最小值；

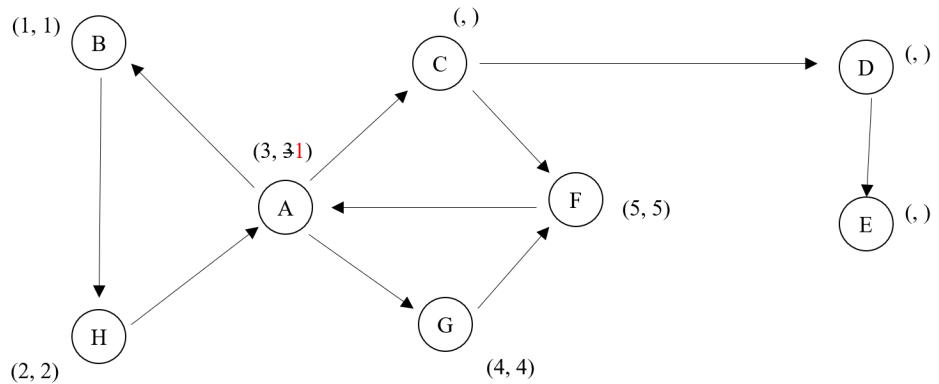
和下家是否在栈里没有关系！如果下家在栈里就pop，不能保证当前顶点还有未遍历的边！！

一个实例：

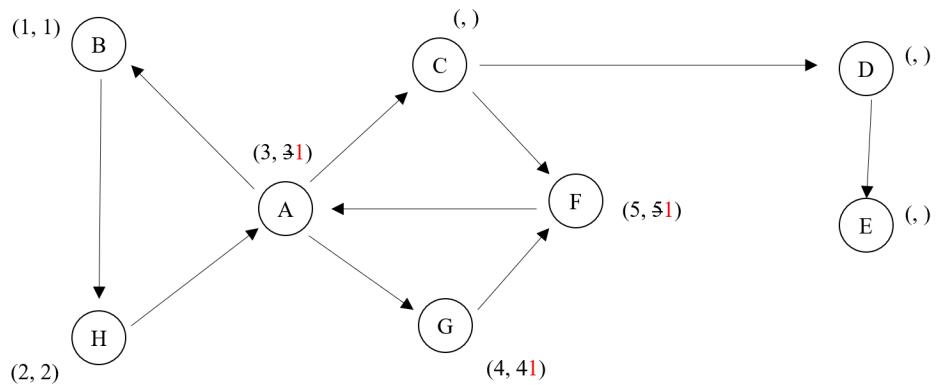
### (Step 1) (dfn, low)



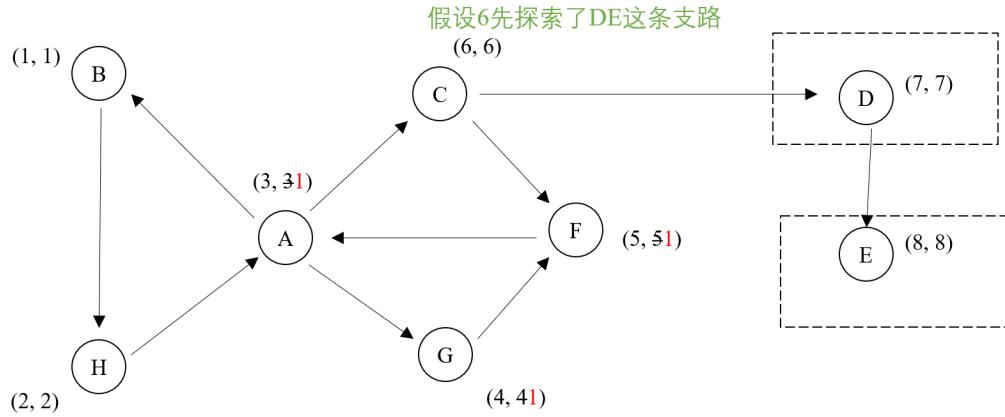
### (Step 2) (dfn, low)



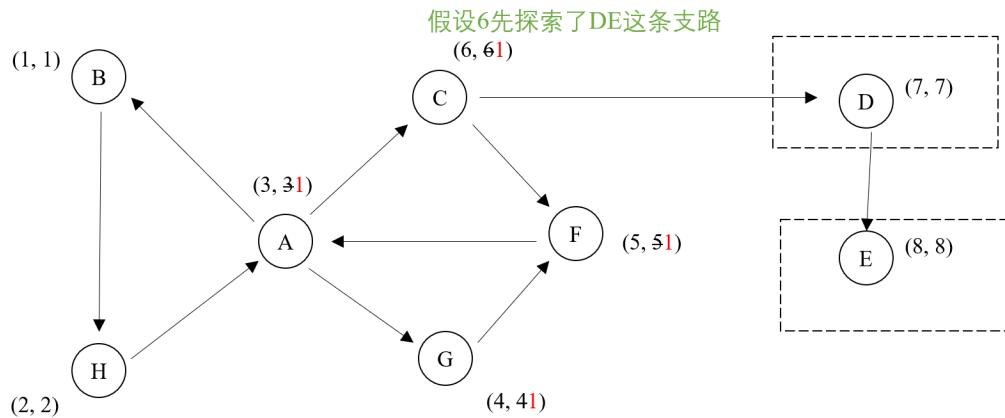
### (Step 3) (dfn, low)



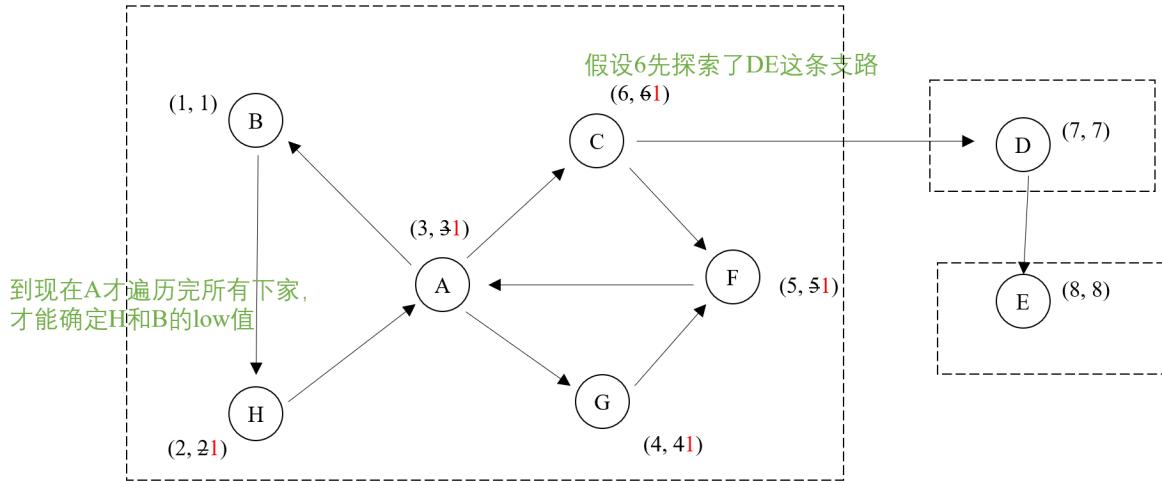
## (Step 4) (dfn, low)



## (Step 5) (dfn, low)



## (Step 6) (dfn, low)



```

1 def dfsTarjan(self, graph):
2
3 for vtx in graph: # 确保不遗漏
4 if not vtx.accessed:
5 self._dfsTarjan(graph, vtx)
6
7 # 先按照 low 排序(同一SCC在一起), 再按照 dfn 排序
8 tmp = sorted(graph, key=lambda x: (x.low, x.dfn))
9 # 将森林按照 dfn==low 的 root: [相同low值的元素] k-v 组成 dict
10 root = None
11 for _ in tmp:
12 if _.low == _.dfn:
13 graph.roots[_] = []
14 root = _
15 else:
16 graph.roots[root].append(_)
17
18 def _dfsTarjan(self, graph, stVtx: Vertex):
19 graph.time += 1
20 stVtx.dfn = graph.time
21 stVtx.low = stVtx.dfn
22 stVtx.accessed = True
23
24 for nextVtx in stVtx.getConnections():
25 if not nextVtx.accessed:
26 nextVtx.pred = stVtx
27 self._dfsTarjan(graph, nextVtx)
28
29 stVtx.low = min(stVtx.low, nextVtx.low)

```

事实上，上面的代码和 Stack 没啥关系……但后果是，后续还需要排序来确定，增加了时间复杂度。

**更好的做法是，利用 Stack 来实现：**

先不pop，而是在每一次递归，遍历完节点 `currVtx` 的下家后：

1. 检查 `currVtx` 的 `low==dfn`；
2. 如果 `True`，说明是这是个 `root`，而在栈里可能还有若干个SCC顶点，堆在根顶点的上面；
3. 那就 `pop`，直到把 `low==dfn` 的退出来；

因为Stack里的顺序是访问的顺序。如果True，说明不可能再访问到前序节点；如果False说明还没确定根，暂时先存在栈里。

只需要在辅助函数最后加上：

```

1 if stVtx.dfn == stVtx.low:
2 tmpLst = []
3 while True:
4 tmpVtx = self.stack.pop()
5 if tmpVtx != stVtx:
6 tmpLst.append(tmpVtx)
7 else:
8 graph.roots[stVtx] = tmpLst
9 break

```

### 7.7.3 割点、割边、公共祖先(LCA, Lowest Common Ancestor): Tarjan算法的其他妙用

**割点(Cut Vertices, Cut Vertex):** 在一个无向图中，如果有一个顶点集合，删除这个顶点集合以及这个集合中所有顶点相关联的边以后，图的连通分量增多，就称这个点集为割点集合。如果去掉一个点以及与它连接的边，该点原来所在的图被分成两部分（不连通），则称该点为割点。

**割边(Bridge, Cut Edge):** 在一个无向图中，如果去掉一条边，该边原来所在的图被分成两部分（不连通），则称该边为割边。

#### 割点

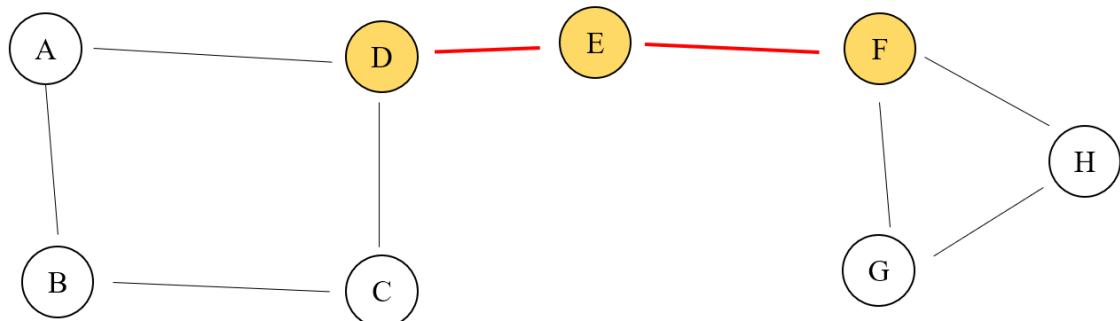
[CSDN上的一个帖子](#)

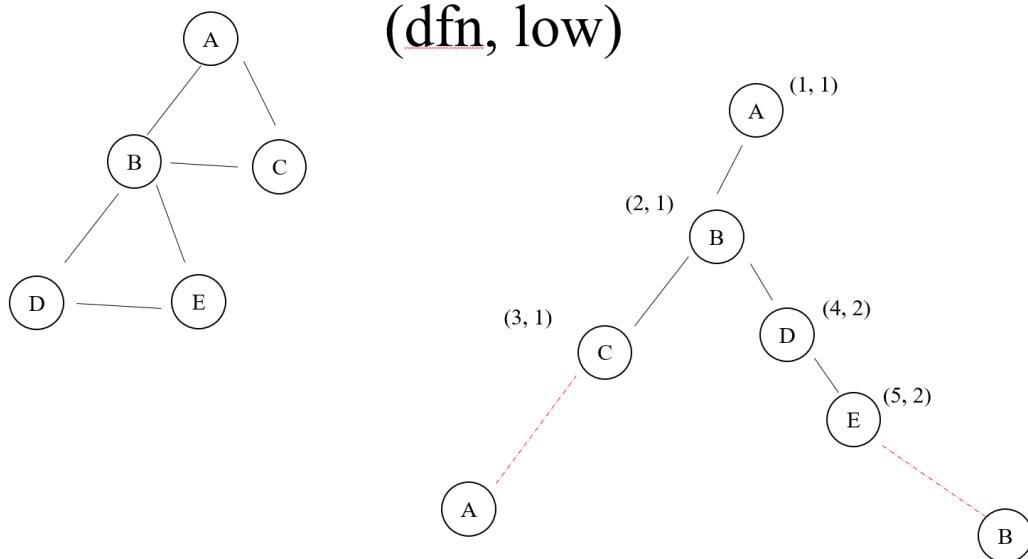
基本想法差不多，都是在DFS的基础上增加时间戳和追溯值。

但SCC的low是和下家的low比较，看的是能走通的顶点：

CV的low是和dfn比较，而且只能向下不能向自己的父节点查询——这个判断是影响low更新的，不要根据节点是否被访问过决定是否递归向下搜索的逻辑混淆了。

以下图为例，以ABCDEFGH为顺序：当D作为当前节点时，只能根据AE来更新low；但是D只能向E递归——这是两个不同的逻辑。





上图说明：E最多只能直接访问到B，而如果要访问比B的时间戳更早的顶点A，就必须要通过B来访问。

假设存在一条通路，从E直达A，那么E的low值就是1，说明E可以不通过B直接走到A。

所以判断割点的条件是：

1. 对于非根节点：如果子节点的low  $\geq$  非根节点的dfn，说明该节点是割点；
2. 对于根节点：如果在树上有超过1条的分支，说明去除这个根节点，就能让分支之间互不连通。

```

1 cnt = 0 # 统计每个顶点在生成树上有几个实际的子节点
2 for next_vtx in curr_vtx:
3
4 if not next_vtx.accessed:
5 next_vtx.pred = curr_vtx
6 cnt += 1
7 _cutVtxTarjan(graph, next_vtx) # 只有下家未被访问过，才需要递归深入
8 curr_vtx.low = min(curr_vtx.low, next_vtx.low) # 下家的值会递归向上传递
9
10 elif next_vtx != curr_vtx.pred: # 对于最下的下家，要么没有子节点，要么就是已访问的顶点
11 # 没有什么下家了，只看相连节点的dfn
12 curr_vtx.low = min(curr_vtx.low, next_vtx.dfn)
13
14 else: # 相连的父节点，没什么可操作的
15 pass
16
17 if curr_vtx.dfn <= next_vtx.low and curr_vtx not in graph.forest:
18 graph.cut_vertices.add(curr_vtx)
19
20 if curr_vtx in graph.forest and cnt >= 2:
21 graph.cut_vertices.add(curr_vtx)

```

2, 割边：对于任意有边连接的点 $u, v$ ，若 $\text{low}[u] > \text{dfn}[v]$ ，则说明边 $u-v$ 为一条割边。

## 7.8 最短路径问题: Dijkstra算法

| 考虑权重了开始

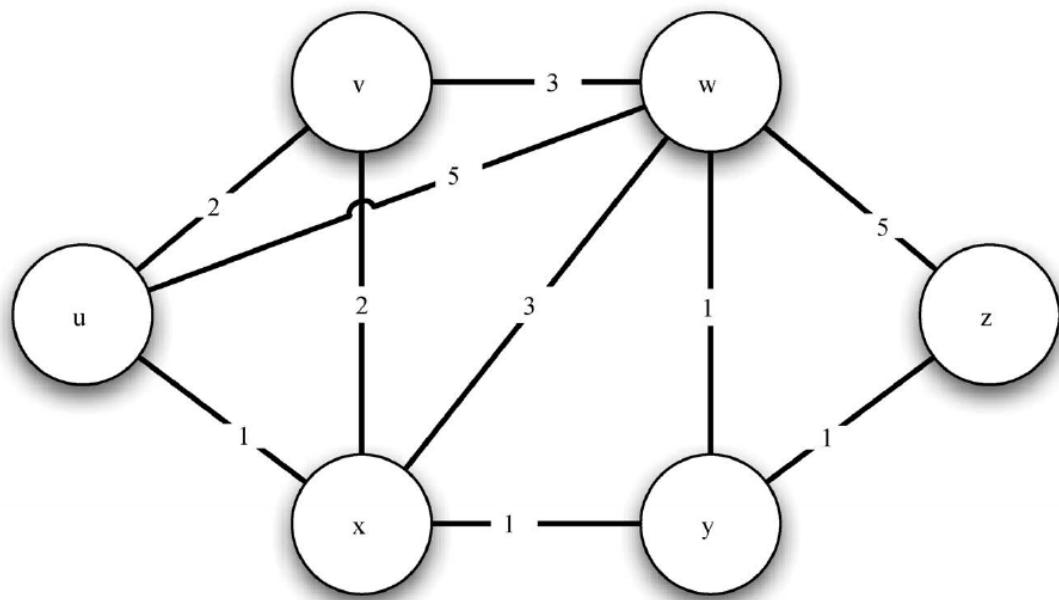


图 7-29 小型路由器网络对应的带权图

一个常用的变体是，指定图中一个顶点，求所有顶点到这个指定顶点的最小距离。

适用条件：**无向，无自环**

思路：

1. 构建一个最小堆，将各个顶点按 `distance` 值排序；`distance` 的物理意义是该节点到指定节点的距离；
2. 将指定顶点弹出来，其 `distance=0`；将其作为**当前顶点**；
3. 对于和**当前节点相连，并且在堆中剩余的顶点**，更新他们的 `distance`：

$$\text{VERTEX}_{\text{distance}} = \min (\text{VERTEX}_{\text{distance}}, \langle \text{VERTEX}, \text{Current} \rangle_{\text{distance}} + \text{Current}_{\text{distance}})$$

4. 也要在堆中更新；
5. 将堆顶顶点 `pop`，作为**当前顶点**；
6. `Goto 3`，直到堆空。

---

在第3步中，每个顶点的距离，要么是和起始点的直线距离，要么是经过一些不在堆里顶点中转的结果。

还可以在更新 distance 的时候顺便更新 pred，方便追查具体的路径。

每次循环只更新和当前节点（最近一次确定最短路径的节点）之间相连的节点。

```
1 def Dijkstra(graph, st_vtx_id):
2
3 # Graph[id] = vertex inst
4 # Graph.__iter__() = get vertices
5 # Vertex[vertex inst] = weight
6 # Vertex.__iter__() = get connections
7
8 # init
9 graph[st_vtx_id].pred = graph[st_vtx_id]
10 graph[st_vtx_id].distance = 0
11 heap = MinBinaryHeapKV().buildHeap([[_distance, _.id] for _ in graph])
12
13 while not heap.isEmpty():
14
15 curr_vtx_id = heap.delMin()[1]
16 curr_vtx = graph[curr_vtx_id]
17
18 for next_inst in (_ for _ in curr_vtx if _.id in heap):
19
20 mid = curr_vtx.distance + curr_vtx[next_inst]
21 if mid < next_inst.distance:
22 next_inst.pred = curr_vtx
23 next_inst.distance = mid
24 heap.replace_key_by_val(next_inst.id, mid)
25
26 pprint([f'{_.id} -> {_.distance} -> {st_vtx_id}, pred is {_.pred.id}'
27 for _ in graph])
```

## 7.9 最小生成树(Minimum Spanning Tree, MST): Prim 算法

**最小生成树，最小权重生成树：**一个有  $n$  个结点的连通图的生成树是原图的极小连通子图，且包含原图中的所有  $n$  个结点，并且有保持图连通的最少的边。

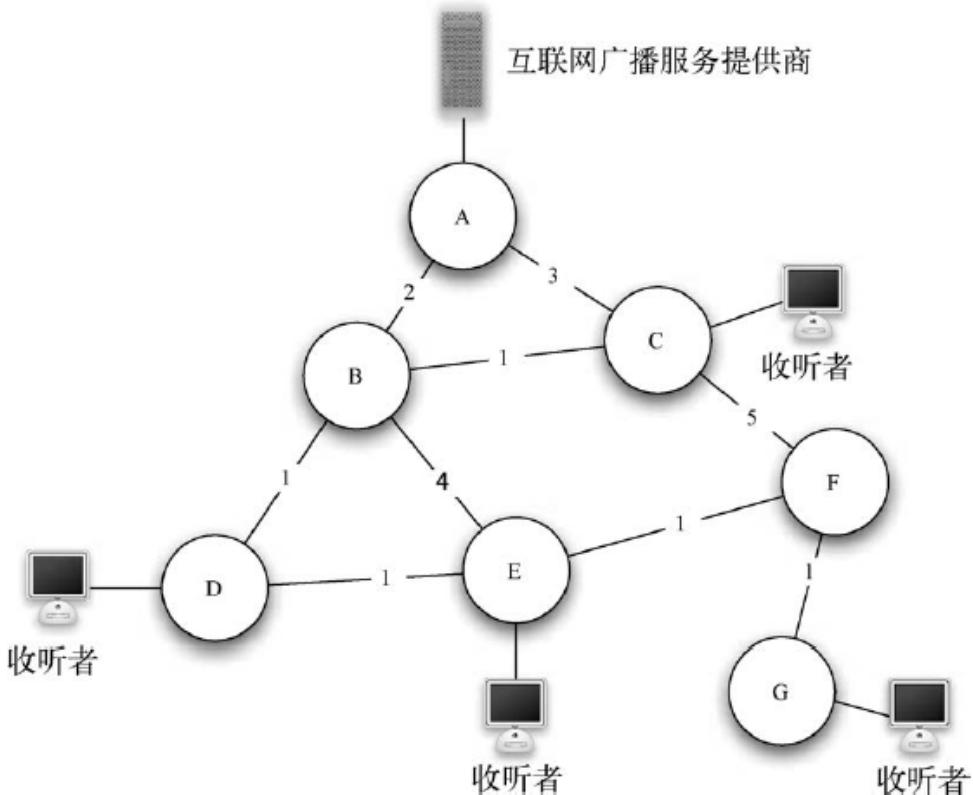


图 7-31 广播问题

**无控制泛滥法：**借助TTL广播，负载很大

**点对点：**为每个用户都发送一条消息，负载也比较大，比如BD会收到三条消息。

如果能构建一颗**最小生成树**的话，每个节点都只收到一条消息，并转发：A2B, B2D&C, D2E, E2F, F2G

算法思想：

可以用GA。从起始点开始，找最近的顶点；再对最近的顶点找最近的顶点——但是这样可能有遗漏。

所以正确的做法是：**确定一个顶点子集，距离顶点子集最近的点下一步被扩充到子集中**。因此每个顶点被扩充到子集中之前，都要和子集中的每个顶点比较距离。

初始值：

1. 所有顶点的 `pred` 为空，`dist` 为一个很大的值，类似于无穷大。
2. 起始点的 `dist` 置0；
3. 构建一个堆 `[vertex.dist, vertex.id]`。

在堆不为空的情况下，循环：

1. `pop` 一个最小顶点作为 `curr`；
2. 对于和 `curr` 相连，并且在堆中的节点 `next`，检查（内循环）：
  1. 新的距离为 `curr` 和 `next` 的距离，加上 `curr` 的距离；

2. 如果新的距离比 next 本身的要小，需要更新 dist=new, pred=curr, heap[dist]。

---

每一个堆中顶点在离开堆之前，都和堆外的每个元素比较过距离。

```
1 def Prim(graph, st_vtx_id):
2
3 # init
4 graph.getVertex(st_vtx_id).distance = 0
5 graph.getVertex(st_vtx_id).pred = graph.getVertex(st_vtx_id)
6 heap = MinBinaryHeapKV().buildHeap([[_distance, _id] for _ in graph])
7
8 while not heap.isEmpty():
9 [curr_dist, curr_id] = heap.delMin()
10
11 # in heap, and connected to heap
12 for next_inst in (_ for _ in
graph.getVertex(curr_id).getConnections() if _.id in heap):
13 new_dist = graph.getVertex(curr_id).getWeight(next_inst) +
curr_dist
14 if new_dist < next_inst.distance:
15 next_inst.distance = new_dist
16 next_inst.pred = graph.getVertex(curr_id)
17 heap.replace_key_by_val(next_inst.id, new_dist)
18
19 pprint([f'{_.id} -> {_.distance} -> {st_vtx_id}, pred is {_.pred.id}'
for _ in graph])
```

## 7.10 Outro

---

对于解决下列问题，图非常有用：

1. 利用宽度优先搜索找到无权重的最短路径。
2. 利用 Dijkstra 算法求解带权重的最短路径。
3. 利用深度优先搜索来探索图。
4. 利用 Kosaraju 算法 或 Tarjan 算法 求强连通单元来简化图。
5. 利用拓扑排序为任务排序。
6. 利用 Prim 算法 生成最小生成树并广播消息。